TECHNISCHE UNIVERSITÄT MÜNCHEN

Lehrstuhl für Sicherheit in der Informatik

Leveraging Derivative Virtual Machine Introspection Methods for Security Applications

Jonas Pfoh

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Arndt Bode

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Claudia Eckert
2. Univ.-Prof. Dr. Uwe Baumgarten

Die Dissertation wurde am 02. Oktober 2012 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 11. Februar 2013 angenommen.

Acknowledgements

This thesis is the result of a great passion and interest in computing security and there have been many that fostered this love and interest over the years. First and foremost, I would like to thank my advisor, mentor, and supervisor, Prof. Dr. Claudia Eckert. Always supportive, enthusiastic, and willing to let me experiment, one could not ask for more in a mentor. I consider myself very lucky to have been extended the opportunity to work under her tutelage. Additionally, I would like to thank Prof. Dr. Uwe Baumgarten for his input and willingness to be a meaningful part of my doctoral committee.

Next, I would like to thank my loving and supportive family. My wife, Dominika, and my wonderful son, Oliver, thank you for your support and love. You make every single day worthwhile. My parents, Armin and Gabriele, and my siblings, Lina and Noah, you have been such a fundamental part of my support structure. I would not be where I am today without you.

There have been so many that share my passion professionally and academically. Having so many people around me that are passionate about the same things has been a blessing. Thank you to my friends and colleagues, Dr. Christoph Krauss, Christian Schneider, Christoph Söllner, Dr. Thomas Stibor, Dr. Frederic Stumpf, and Sebastian Vogl. Your input and perspective has always been extremely invaluable. I always benefited from the, often passionate, discussions that I had with each one of you. Finally, I would like to thank those that helped me tend my "other baby", Nitro, through their contribution to the project, Julian Kirsch, Christian Fuchs, Cornelius Diekmann, and Marcus Fensterer.

Abstract

Virtual machine introspection (VMI) describes the method of monitoring, analyzing, and manipulating the state of a virtual machine from the hypervisor level. This lends itself to many security applications, though they all share a single fundamental challenge: One must address the fact that the hypervisor has no semantic knowledge about what the system state means (e. g., where key data structures are). Traditionally, this semantic knowledge is simply delivered to the hypervisor in the form of debugging symbols, symbol tables, etc. If such semantic information does not exist, it must be acquired through other, often tedious, means such as reverse engineering or "trial and error". On the other hand, it is possible to derive information about the guest OS by considering hardware features and their specifications. This derivative method is possible without any delivered semantic information about the guest OS and has several additional advantages including guest OS portability and resistance to evasion techniques.

The main contribution of this research is an examination of derivative VMI methods and their strengths. As there is little prior work formally exploring the potential of derivative VMI methods, we inspect Intel's IA-32 and IA-32e architectures and investigate their potential for derivative VMI. Through this inspection, we discover and present several portions of the hardware specifications which are conducive to derivative VMI.

This culminates in the introduction of a novel derivative method for collecting system calls from the hypervisor. This method is completely guest OS agnostic and has been tested on a variety of guest OSs. In addition, our method cannot be evaded from within the guest. Furthermore, we show that our method keeps the collection overhead to a minimum by comparing its performance to a similar system. We extend this work by presenting a novel approach to malware detection that makes use of a support vector machine (SVM) to classify system call traces. In contrast to other methods that use system call traces for malware detection, our approach employs a string kernel to make better use of the sequential information inherent in a system call trace. By classifying system call traces in small sections and keeping a moving average over the probability estimates produced by the SVM, our approach is capable of detecting malicious behavior online and achieves a very high accuracy.

Zusammenfassung

Virtual-Machine-Introspection (VMI) bezeichnet die Überwachung, Analyse und Manipulation des Zustands einer virtuellen Maschine auf der Hypervisor-Ebene. Diese Methode eignet sich besonders für Sicherheitsanwendungen. Eine fundamentale Herausforderung dieses Ansatzes ist das Fehlen von semantischem Wissen über den Systemzustand innerhalb des Hypervisors (z. B. wo im Speicher sich wichtige Datenstrukturen befinden). Traditionelle Methoden liefern dieses semantische Wissen einfach an den Hypervisor in Form von Debugging-Symbolen, Symboltabellen usw. Wenn dieses semantische Wissen nicht vorhanden ist, muss es durch andere, oft mühsame Wege wie Reverse Engineering erworben werden. Anderseits ist es möglich, Wissen über den Zustand des Gast-Betriebssystem aus den Hardwarefunktionen und ihren Spezifikationen abzuleiten. Diese, sogenannte *derivative Methode* ist ohne zusätzliches semantisches Wissen über das Gastsystem möglich und hat mehrere zusätzliche Vorteile, beispielsweise Portabilität zwischen verschieden Gastsystemen und Resistenz gegen Umgehung der Isolation und der Sicherheitsmechanismen.

Der wichtigste Beitrag dieser Forschungsarbeit ist die Untersuchung von derivativen VMI-Methoden und ihrer Stärken. Da es bisher wenige Arbeiten in dieser Richtung gibt, untersuchen wir die IA-32 und IA-32e Architekturen von Intel sowie deren Potenzial für derivative VMI-Techniken. Durch diese Untersuchung haben wir diejenigen Teile der Hardware-Spezifikationen identifiziert, die geeignet für derivatives VMI sind.

Diese Untersuchung führt letztendlich zu der Einführung eines neuen derivativen Verfahrens zur Erfassung von System-Calls durch den Hypervisor. Diese Methode ist vollständig portabel zwischen verschieden Gast-Betriebssystemen und ist mit einer Vielzahl von Systemen getestet worden. Ein besonderes Merkmal unserer Methode ist, dass der Mechanismus nicht vom Gast umgegangen werden kann. Durch einen direkten Vergleich mit einem ähnlichen System mit unterschiedlichem Ansatz zeigt sich, dass unsere Methode performant ist und den zusätzlichen Overhead minimiert. Wir verwenden die erfassten Daten zur Malware-Erkennung. Hierfür präsentieren wir einen neuartigen Ansatz zur Klassifikation von Programmverhalten auf Basis der System-Call-Traces mit Hilfe einer Support-Vector-Machine (SVM). Im Gegensatz zu anderen Methoden, die System-Call-Traces für Malware-Erkennung nutzen, verwendet wir einen String Kernel, um die sequentiellen Informationen besser nutzen zu können. Durch die Aufteilung der Traces in kleine Abschnitte und ein gleitendes Mittel über die Wahrscheinlichkeiten ist unser Ansatz geeignet, bösartiges Verhalten mit sehr hoher Genauigkeit zu erkennen.

Contents

List of Figures

List of Tables

Publications

Jonas Pfoh, Christian Schneider, and Claudia Eckert. Leveraging string kernels for malware detection. In *The International Conference on Network and System Security (NSS)*. Springer, 2013.

Jonas Pfoh, Christian Schneider, and Claudia Eckert. Nitro: Hardware-based system call tracing for virtual machines. In *Advances in Information and Computer Security*, volume 7038 of *Lecture Notes in Computer Science*, pages 96–112. Springer, November 2011.

Jonas Pfoh, Christian Schneider, and Claudia Eckert. Exploiting the x86 architecture to derive virtual machine state information. In *Proceedings of the Fourth International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2010)*, pages 166–175, Venice, Italy, July 2010. IEEE Computer Society. Best Paper Award.

Jonas Pfoh, Christian Schneider, and Claudia Eckert. A formal model for virtual machine introspection. In *Proceedings of the 2nd Workshop on Virtual Machine Security (VMSec '09)*, pages 1–10, Chicago, Illinois, USA, November 2009. ACM Press.

Chapter 1 Introduction

As virtualization has become more prevalent in the recent past, there has been an emphasis on leveraging this technology for security applications. Virtualization generally describes a technology in which an instruction set or binary interface is virtualized. This has many benefits including the ability to emulate foreign hardware or exposing several virtual hardware interfaces on a single physical host. The software layer that exposes this virtual hardware interface (i. e., virtual machine) is referred to as a virtual machine monitor (VMM) or hypervisor[1] and the software that runs within the virtual machine is referred to as the guest.

In order to leverage this technology for security applications it is necessary to inspect or manipulate the guest from the hypervisor at run time. This inspection and/or manipulation is generally referred to as virtual machine introspection (VMI) [26]. A more detailed explanation of virtualization and VMI can be found in Chapters 2 and 3.

1.1 Motivation

Before a hypervisor can perform any sort of introspection of the guest operating system's (OS) state, it must have some semantic knowledge about that guest's state. That is, the introspecting component must understand what the binary state it is examining represents. Without additional knowledge it is difficult to determine which instructions or data structures one is inspecting when simply given binary information. This dissociation between the binary data and its meaning is referred to as the *semantic gap* [14]. The semantic gap is an issue that all VMI mechanisms must address as any component within the hypervisor has no native knowledge about the guest. Addressing this lack of knowledge is a fundamental challenge of VMI.

Traditionally, hypervisors make use of information delivered to them in an out-of-band manner to bridge the semantic gap. For example, the hypervisor is provided with

[1]We use the terms hypervisor and VMM interchangeably.

debugging symbols for the guest OS's kernel before introspection begins. This delivered information provides the hypervisor with the information it needs to interpret the raw state of the guest and perform introspection. This approach, while simple, has two fundamental issues. First, it requires that semantic information be gathered for every potential guest configuration one is interested in inspecting or manipulating. For example, the debugging symbols gathered for Linux kernel version 3.0 may not be applicable for Linux kernel version 2.6.32 and will certainly not be applicable for the Windows XP SP2 kernel. This results in one having to gather this information for each potential guest OS. Furthermore, if the guest OS is closed source, gathering this semantic information becomes very difficult without intricate knowledge of its internals. The second fundamental issue is that the semantic information that the hypervisor is using to perform introspection is not bound to the actual software architecture. That is, the software architecture may dynamically change at run-time, however the semantic information that the hypervisor has remains static. This change in the software architecture may happen by design in the form of an update or for malicious reasons. In any case, if the hypervisor is not updated with new semantic information it may no longer have a correct understanding of what is happening within the guest. This may be especially dangerous in a security application.

There is an additional method for bridging the semantic gap, namely the derivative method. This method addresses the semantic gap by looking to the (virtual) hardware. That is, much can be derived about an OS simply by understanding the hardware on which it runs. The hardware operates under a strict set of specifications that all software must adhere to. For example, the Intel IA-32 architecture specifies that the CR3 register must hold the address of the top-level page directory when paging is active [35]. All OSs that wish to run on this hardware must adhere to this specification because if they do not, processes will not have access to their individual address spaces. It is this observation that the derivative method takes advantage of. That is, with a thorough knowledge of the hardware specifications, information can be derived about the guest OS. If we continue with our example, a VMI mechanism taking a derivative approach might make use of this knowledge of how the paging system works to enumerate processes within the guest without any information about the guest OS itself. Clearly, this approach does not suffer from the same issues that the delivery approach suffers. Any guest OS that wishes to run on the hardware, must abide by the specifications and therefore a derivative approach will work for all guests that run on a given hardware without additional effort. Additionally, the hardware architecture will not change against specification at run-time. Therefore we can be assured that the information we are basing our introspection on will remain valid throughout the lifetime of the guest.

While this approach seems promising one must explore to what extent it is feasible. Clearly, there will be portions of the guest state that require delivered information to understand. There are OS mechanisms that do not require special assistance from the hardware. It will be difficult, if not impossible, to derive information about these mechanisms. This means that one must take the time to understand the hardware and identify what useful information can realistically be derived about the guest OS. Additionally, it must be shown that such derivative approaches can be effective from a security perspective and one must explore what impact they have on performance.

1.2 Contribution

This thesis attempts to address the question, “What information can we derive about a guest OS, given that we know nothing about the guest OS?”. From this follows, “What feasible properties can we identify that hold for derivative methods of VMI?”, “Is the gathered information useful from a security perspective?”, and “Can we gather this information in a efficient manner?”. Concretely, the contributions of this thesis are summarized in the following paragraphs.

Identifying the Derivative Method as a Feasible Approach to VMI

We begin our work by exploring the challenges that must be overcome when leveraging VMI to its full potential and we introduce a taxonomy which helps in examining and reasoning about possible VMI approaches. We introduce several patterns to describe the various methods in which VMI components bridge the semantic gap. As part of this taxonomy, we describe both delivery and derivative approaches and draw conclusions about the advantages and disadvantages of each. Finally, we conclude that a derivative approach has only been used in limited cases and has properties that make it interesting for further research.

Identifying Portions of Intel’s IA-32 and IA-32e Architectures Viable for Derivative Mechanisms

After identifying derivative methods as the target for further research, we take a general approach and look at what the hardware specifications for Intel’s IA-32 and IA-32e architectures[2] provide. We pay special attention to the virtualization extensions as well as to those mechanisms that show promise for a derivative approach. Specifically, we look at the specifications for various processor registers and discuss how they are relevant to derivative VMI. In addition, we examine the virtualization extensions that Intel provides and identify where they are lacking from a VMI perspective. That is, the virtualization extensions were designed with performance in mind and not VMI. This leaves us in a situation where there are often events that we wish to trap to the hypervisor, however trapping these events is not supported by the hardware. For these situations, we discuss how the interrupt mechanism may be used to address these situations.

[2]We will collectively refer to these two architectures as the x86 architecture going forth.

Identifying Feasible Properties for VMI-based Security Mechanisms

In addition to the properties native to derivation-based VMI mechanisms, we explore further properties for VMI-based security mechanisms. Particularly, we take a close look at the property of transparency as it relates to hypervisors as well as VMI mechanisms. In 2007, Garfinkel et al. considered hypervisor transparency and concluded that "...building a transparent VMM is fundamentally infeasible, as well as impractical from a performance and engineering standpoint" [24]. They argue that this is due to discrepancies in the VM that, from an engineering standpoint, will always remain in some fashion. These discrepancies include simple logical discrepancies, such as instructions that do not perfectly conform to the specification, as well as timing discrepancies that will always remain as long as the hypervisor must perform some non-virtualizable actions in software. We extend this argument to include VMI mechanisms and consider the drivers for the need for transparency. With these drivers identified, we propose feasible properties that can displace transparency in certain situations.

Presenting an Introspection Mechanism That Makes Use of Derivative Methods

We provide a proof of concept that derivative methods are feasible for useful VMI mechanisms. We do this in the form of Nitro, a framework for trapping system calls to the hypervisor in a derivative manner. We show that Nitro is capable, useful, flexible, and resistant to evasion attempts.

Demonstrating that Nitro is Both Efficient and Useful from a Security Perspective

Finally, we consider the performance of Nitro by looking to the methods used to facilitate system call trapping. We consider some of the design decisions we made and argue as to why these work to boost performance. In addition to some standalone performance testing, we also compare Nitro to a similar system that relies on delivered information and show that Nitro out performs this system. The final step we take is to consider the system call traces themselves and show that these can be used to successfully perform malware detection. To do this, we take advantage of support vector machines (SVMs) in conjunction with string kernels, initially introduced for natural language processing.

1.3 Outline

The second chapter of this thesis serves to provide general background to virtualization. Chapter 3 expands on this by taking a closer look at VMI. This includes a taxonomy that we introduce to help in describing and reasoning about VMI approaches. We follow this with a short look at related work and identify that there has been relatively little research in the area of derivative VMI methods. We conclude Chapter 4 by arguing that derivative VMI mechanisms are worthwhile for further research based on the properties they inherently provide in addition to the limited amount of existing work. Chapter 5 begins with a look at transparency and how it relates to virtualization in general and VMI specifically. After arguing that transparency is an infeasible goal, we investigate the drivers for transparency in a VMI application and present feasible goals in lieu of transparency. We conclude Chapter 5 with requirements to achieve each goal. This is followed by an in-depth look at Intel's x86 architecture in Chapter 6. This chapter aims to describe general approaches to derivative VMI mechanisms as well as the specific features of the x86 architecture that lend themselves to derivative methods. The seventh chapter introduces Nitro, our framework for trapping system calls to the hypervisor using derivative methods. In this chapter we describe the implementation of Nitro and include a look back at the properties discussed in Chapters 3 and 4. We demonstrate that Nitro, in fact, carries these properties. Chapter 8 provides an evaluation of Nitro which includes the results of performance testing in addition to a comparison with another system for system call trapping from the hypervisor. Chapter 9 aims to take the system call traces produced with Nitro and show how they can be used to effectively detect malware using machine learning techniques. Specifically, we look at support vector machines (SVMs) in conjunction with string kernels. String kernels are interesting as they have never been used in malware detection, to our knowledge, and have interesting properties that make them particularly suitable for malware detection. Chapter 10 presents the results of our experimentation using our SVM-based approach to malware detection. In addition, we compare our method to several other machine learning methods for malware detection using system call traces and show that our approach performs among the best. Finally, we conclude this thesis in Chapter 11.

Chapter 2

Background

This Chapter will detail the groundwork that this thesis is built upon. Since virtualization covers a very broad spectrum, we will acquaint the reader with the necessary background.

2.1 Virtual Machines

In order to understand virtualization, it is necessary to understand what a virtual machine (VM) is. Furthermore, in order to understand what a virtual machine is, we must define a machine. A *machine* is a system that performs useful work on one's behalf. In order to perform this work, it is often necessary to provide input. In this case, the machine must provide an interface for accepting this input. For example, computing hardware is a machine that provides an interface in the form of registers and the instructions it accepts. Generally, an OS will make use of this interface to have the hardware perform work on its behalf. This work might consist of simple arithmetic operations or more complex I/O. If we expand this example further, the OS together with parts of the hardware may also be considered a machine that does useful work on behalf of a process. This is a layered architecture with the hardware representing the bottom-most layer as is depicted in Figure 2.1

The interface that is provided by a machine plays an important role. In order to ease discussion, we introduce names for the various interfaces. The interface that the hardware provides is referred to as the *instruction set architecture* (ISA) [75] interface. The ISA is, depending on the hardware, generally quite complex and can be further divided into two parts. First, the user ISA is that portion of the ISA that any layer above may access. In contrast, the system ISA is that portion of the ISA which facilitates access to specific system resources and only privileged layers above may make use of. This privileged layer generally consists of the OS. In Figure 2.1, interface '1' depicts the system ISA, while interface '2' depicts the user ISA and together they form the ISA.

The interface that the OS and the user ISA collectively provide to the layers above

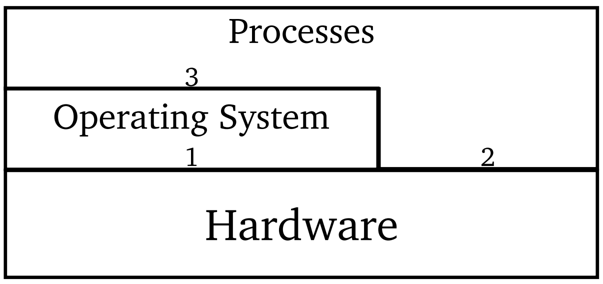


Figure 2.1: Computing system layered architecture

is referred to as the *application binary interface* (ABI) [75]. While a process may not have direct access to the system ISA, it may still need to access system resources. To facilitate this, the OS provides an interface to the layers above, generally in the form of *system calls* (Figure 2.1, interface '3'). These system calls allow a process to use system resources while allowing the OS to control that access. In Figure 2.1, the ABI is depicted by interfaces '2' and '3'.

Given these definitions, we can better understand what a virtual machine is. A virtual machine implements (i. e., virtualizes) a machine in software. This leads to several options. For example, one can implement a virtual machine that provides a virtual ISA interface for a system to run on. Additionally, one can implement a virtual machine that provides an ABI directly for processes. Specifically, we refer to a virtual machine that provides a virtual ISA interface for an entire system as a *system VM* and we refer to a virtual machine that provides an ABI for a single process as a *process VM* [75].

2.2 Process Virtual Machines

In the purest sense, many multitasking OSs are, by their nature, a process VM. They provide an ABI, which is implemented using the ISA that the OS is running on, and provide each process an individual address space. That is, the process may run as if it is the only process running in memory. However, generally when we speak of a process VM, we refer to an additional layer between the OS and the process.

A process VM must implement the ABI that it exports (we will refer to this interface as the *target interface*) using the interface on which the VM is running (we will refer to this interface as the *host interface*). That is, if a process VM is running directly on an OS, the VM must implement the target interface that it provides to layers above using the host interface that the underlying OS provides. In order to accomplish this, the VM may rely on one of two methods, either interpretation or binary translation [75].

Interpretation is the most straightforward method in that it simply relies on emulating each individual instruction accepted by its target interface using instructions accepted by the host interface. While this method may be straightforward, it may also prove inefficient, as each target instruction may require several instructions on the host interface.

On the other hand, *binary translation* attempts to address the inefficient manner of interpretation. This method attempts to inspect the process and translate blocks of target instructions into blocks of instructions for the host interface that perform the same task [74]. Once this translation is complete, the translated block may be run directly on the host interface. While the initial translation of a block may incur high time overhead, subsequent executions of that block will not incur the same overhead as these translations may be cached.

It is also possible to combine these two methods in an effort to streamline the VM. That is, one may use interpretation for portions that may not be particularly performance critical as it is the more straightforward method and use binary translation for portions that are performance critical.

One popular advantage of process VMs is that they can be implemented in such a way that they provide a platform-independent environment for their processes. That is, one must put in the effort of implementing the VM for several platforms. However once this is finished, processes written to run within the VM can run on several platforms without adjustment. This approach can be seen in many popular examples such as the Java VM and the ActionScript VM (Adobe Flash).

2.3 System Virtual Machines

The term *virtual machine* was coined to describe system virtual machines during their inception in the 1960s and 1970s [29, 75]. A system virtual machine provides an ISA interface for an entire system to run on. The system running within the VM is often referred to as the *guest OS* while the software layer that manages the VM is often referred to as the *virtual machine monitor* (VMM) or *hypervisor*[1]. These components are shown in Figure 2.2.

System VMs were originally conceived and used in the 1960s and 1970s, but lost their acclaim in the 1980s and 1990s due primarily to reduced hardware costs. That is, as hardware became smaller and cheaper, it also became more accessible and there was less need to share hardware through virtualization. However, starting in the 2000s there has been a resurgence of virtualization technology both in industry and in academia [67]. This has several reasons. The first is, once again, cost. Although the cost of hardware has come down since the 1970s, the proliferation of computing technology in all aspects of society has, once again, made virtualization an attractive option to reduce costs and to manage space. In addition to cost, the isolation that system virtualization offers is an attractive property in a variety of applications. System virtualization is a straightforward mechanism to mitigate the consequences of crashes or intrusions. Looking to the future,

[1]We use the terms hypervisor and VMM interchangeably.

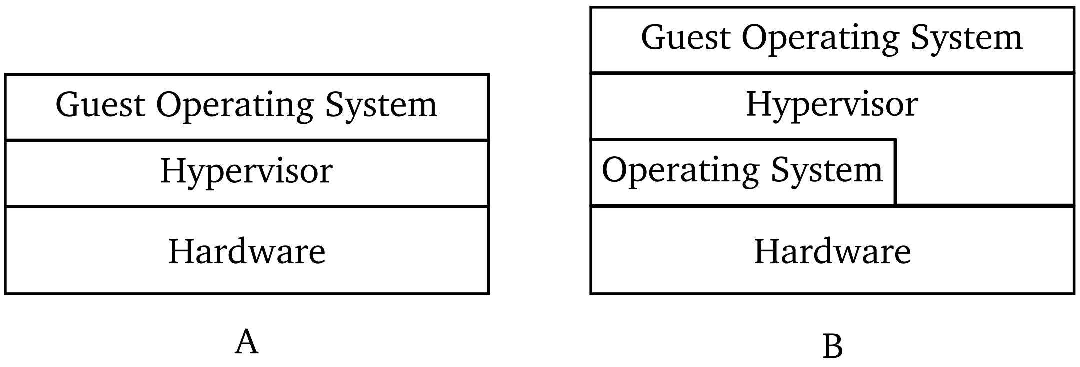


Figure 2.2: Bare-metal VM (Type I) architecture (A) and Hosted VM (Type II) architecture (B)

Rosenblum and Garfinkel speculate that system virtualization will be an increasingly sought after technology to address security and reliability [67].

2.3.1 System Virtual Machine Architectures

While system VM architectures vary greatly, they can be broadly divided into two categories: bare-metal VMs (Type I) and hosted VMs (Type II) [28].

In bare-metal VM architectures, the VMM runs directly on the hardware as is depicted in Figure 2.2(A). In contrast, a hosted VM runs within an OS environment as shown in Figure 2.2(B). While a bare-metal VM architecture is generally less complex due to the smaller code base which results from the lacking host OS, the hypervisor cannot rely on a host OS to provide low level services and must implement them on its own. For example, in a bare-metal VM architecture, the VMM must include drivers for all hardware devices. Whereas in a hosted VM architecture, the VMM can rely on the host OS to provide access to the hardware. While the VMM can rely on the host OS for some services, it may still be beneficial for the VMM to access portions of the hardware's system ISA. For this reason, it is often the case that, even in a hosted VM architecture, a portion of the VMM is integrated into the host OS in the form of drivers or modules.

2.3.2 System Virtual Machine Interfaces

System VMs may be roughly categorized based on the virtual ISA interface (i. e., target interface) they offer. The first of such categories is an emulation VM. That is, the VM offers an ISA that may differ from the underlying hardware's ISA. This allows a system intended for a particular hardware architecture to run on unintended hardware. Traditionally, VMs and emulators were often acknowledged independently, however they share many concepts [52]. As a result of this, more current definitions of VMs include emulation VMs [75]. As with process VMs, emulation can be implemented either through interpretation or binary translation. Interpretation is a matter of directly implementing the target interface in software and interpreting each target instruction for the host architecture. In contrast, binary translation relies on translating blocks of target instructions to functionally equivalent host instructions. While interpretation incurs a relatively high performance overhead, its strength lies in its flexibility. There are no interfaces which cannot be emulated through interpretation. That is, emulating an architecture by means of interpretation allows one to emulate any architecture, even architectures for which hardware may not yet have been released [51]. Conversely, while binary translation may decrease the performance overhead, it may not be able to emulate all interfaces [75]. For example, if the target architecture supports a mechanism such as virtual addressing and the underlying host architecture does not offer such a mechanism, there might not be a way to translate this and the virtual address mechanism must be implemented in software (i. e., interpreted).

In direct contrast to emulation, a same-ISA VM offers an ISA that is identical to that of the underlying hardware. If the host ISA supports various privilege levels, these can be leveraged to implement such a VM. If the reader will recall from Section 2.1, the ISA may be divided into two parts, namely the user ISA and the system ISA. In this case, the VMM runs with system privileges and may make use of the entire host ISA, while the guest OS runs with user privileges and may make direct use of the host user ISA. Since the guest OS will still try to access the host system ISA interface, the VM must take care to virtualize this portion of the ISA for the guest OS [29, 59, 75]. Popek and Goldberg identified three properties that a same-ISA system VM should possess, namely efficiency, resource control, and equivalence [59]. They stress that user ISA instructions must be executed natively, the hypervisor must have full control over all resources, and that programs running within such a VM should perform identically to the same program running on physical hardware. These VMs incur much less time overhead compared to emulation. Such an approach can be improved further through special hardware assistance for virtualization in the physical hardware. This is explained further in Section 2.3.3.

Between emulation and same-ISA VMs, there has emerged an approach that lies somewhere between these two more traditional methods of system virtualization, namely paravirtualization [89, 88]. In a paravirtualized environment the VM offers a target ISA interface that consists of the host ISA interface (or a majority thereof) in addition to some extended interface, sometimes referred to as the *hypercall interface* [9]. Generally, the portion of the VM that offers the host ISA interface is implemented in the same manner as a same-ISA VM, while the hypercall interface is implemented through interpretation. These hypercalls are generally very few and allow the guest OS to send status signals to the VMM or make special requests of the VMM that either the physical hardware does not support or streamlines the sharing of hardware resources. Of course, in such an environment the guest OS must be aware of the hypercall interface in order to make use of it. That is, to take full advantage of paravirtualization a specialized guest OS kernel is required.

2.3.3 Hardware-Assisted System Virtual Machines

As mentioned in Section 2.3.2, same-ISA VM's performance may be improved by making use of hardware extensions that are available in modern processors. While such extensions may exist on other hardware, for the purposes of this discussion we will focus on the extensions provided by Intel (Vanderpool/VT-x) [35] and AMD (Pacifica/SVM) [5] for the x86 family of processors. At a high level, these virtualization extensions add an additional layer to the protection mechanism already available in the x86 architecture.

As discussed in Section 2.1, a hardware platform's ISA may be divided into user and system ISAs. In the x86 architecture, the ISA is split into four privilege levels or rings numbered 0 to 3, where ring 0 provides the highest privilege level and ring 3 provides the lowest. As mentioned in Section 2.3.2, same-ISA VMs may be implemented taking advantage of these privilege levels, in which the VMM runs in ring 0 and the guest in a lower privilege level. The problem with such an implementation is that the guest OS, which expects to run in ring 0, must run at a lower privilege level and the VMM must handle all interaction with the system ISA interface on behalf of the guest. What the virtualization extensions provide is another level of protection. That is, a new set of virtualization privilege modes is introduced, namely VM mode and root mode. Each of these new privilege modes has the traditional privilege levels nested within. So, a guest OS may run within ring 0 as it expects, but may not realized that it is also running in VM mode. There are still some portions of the ISA interface for which the VMM will need to get involved and when the guest attempts to access these, the hardware will switch back to root mode and let the VMM resume. This is referred to as a *VM exit*, while a switch from root mode to VM mode is referred to as a *VM entry*.

This allows the guest OS to access those parts of the system ISA interface that are not critical for the purposes of virtualization, thus improving performance as the guest OS is running directly on the host hardware. The VMM must only get involved in those portions of the system ISA interface that are critical for virtualization.

2.4 Conclusion

The background in this chapter lays the foundation for an interesting concept that arises in virtualized environments, namely virtual machine introspection (VMI). As system virtualization inherently gives the hypervisor complete control of all system state information, the hypervisor can use this knowledge to inspect, interrupt, and/or manipulate the VM. This has very interesting implications especially from a security perspective. In the next two chapters we explore VMI more extensively. We will explore the VMI process, introduce patterns by which one can perform VMI, and discuss common challenges and properties of VMI.

Chapter 3

Virtual Machine Introspection

Virtual machine introspection (VMI) describes the act of examining, monitoring, and manipulating the state of a guest OS running inside a virtual machine from the isolation of the hypervisor.

The above definition of virtual machine introspection is based on the foundation that Garfinkel and Rosenblum laid [26]. They outline three properties of virtualization that enable VMI:

Isolation This property ensures that the guest OS remains in confinement and is unable to tamper with the VMM or any component within.

Inspection This property allows the VMM to examine the entire state of the guest OS.

Interposition This property refers to the ability to inject actions into the normal operation of the guest system.

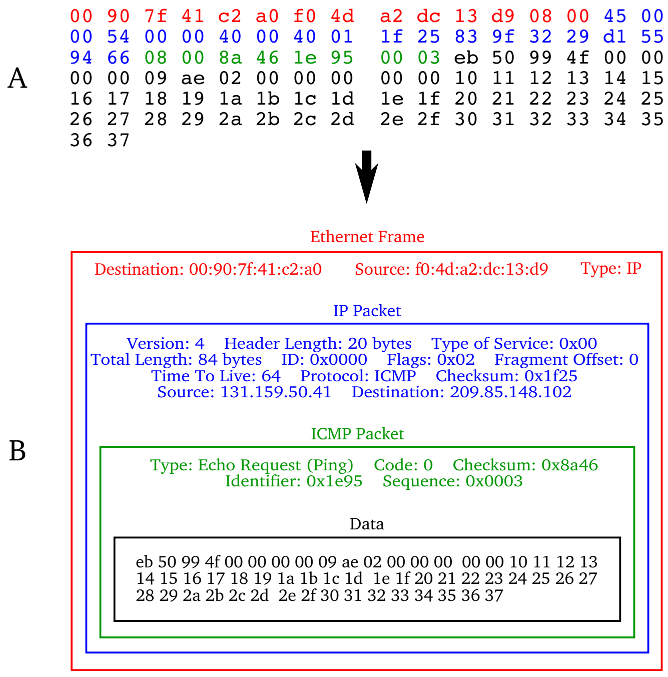
It is from these properties that we developed our definition. The ability to examine and monitor the guest OS is possible through the inspection property. The ability to manipulate the guest OS comes from the interposition property. Finally, the isolation property enforces that this can take place in confinement from the guest OS. Ultimately, these properties allow us to gather a *complete* and *untainted* view of the VM state, operate in complete *isolation* of any entity running within the VM, and *manipulate* the VM state if or when necessary.

3.1 The Semantic Gap

Despite the power that one has from the vantage point of the hypervisor, there remains one fundamental challenge that *all* VMI approaches share:

Interpreting the immense amount of binary low-level data that comprise the system state to obtain an abstract high-level view of the guest OS

This challenge expresses a need to interpret a massive amount of data that, from the vantage point of the hypervisor, is simply binary data. It may be very difficult for a VMI component to act directly on this binary data. This is due to the fact that it is generally an immense amount of data that has little meaning at face value. This lack of meaning or semantics is referred to as the *semantic gap* [14]. The meaning of this binary data must be evaluated such that the VMI component can act upon it. In order to perform this evaluation it is often the case that some external knowledge of the guest hardware or OS is necessary. Once some external knowledge has been applied, it becomes clear what the binary data represents. This process is illustrated in Figure 3.1. Figure 3.1(A) shows the data in its raw state. It may be difficult to incorporate this information into an analysis without additional information. However, when we combine this binary information with further semantic information, we are able to interpret the information and make better use of it in our analysis. In this case, we combine the binary data with the knowledge that it represents an Ethernet frame and the knowledge of the involved protocol specifications to get the view depicted by Figure 3.1(B).



```
00 90 7f 41 c2 a0 f0 4d  a2 dc 13 d9 08 00 45 00
00 54 00 00 40 00 40 01  1f 25 83 9f 32 29 d1 55
94 66 08 00 8a 46 1e 95  00 03 eb 50 99 4f 00 00
00 00 09 ae 02 00 00 00  00 00 10 11 12 13 14 15
16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25
26 27 28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35
36 37
```

Ethernet Frame

Destination: 00:90:7f:41:c2:a0   Source: f0:4d:a2:dc:13:d9   Type: IP

IP Packet

Version: 4   Header Length: 20 bytes   Type of Service: 0x00
Total Length: 84 bytes   ID: 0x0000   Flags: 0x02   Fragment Offset: 0
Time To Live: 64   Protocol: ICMP   Checksum: 0x1f25
Source: 131.159.50.41   Destination: 209.85.148.102

ICMP Packet

Type: Echo Request (Ping)   Code: 0   Checksum: 0x8a46
Identifier: 0x1e95   Sequence: 0x0003

Data

```
eb 50 99 4f 00 00 00 00 09 ae 02 00 00 00  00 00 10 11 12 13
14 15 16 17 18 19 1a 1b 1c 1d  1e 1f 20 21 22 23 24 25 26 27
28 29 2a 2b 2c 2d  2e 2f 30 31 32 33 34 35 36 37
```

Figure 3.1: The binary representation of an ICMP packet (A) is difficult to interpret until semantic information is applied (B).

3.2 The VMI Process

In this section, we take a closer look at how VMI is performed in a general sense. That is, we look at each individual step that is taken starting with collecting the raw state and finishing with a comprehensible aggregated view of what is happening with the guest VM.

3.2.1 States and Views

All VMI approaches inspect guest system state. The guest system state is comprised of all guest system information and is simply the concatenation of all observable values within the VM. These observable values include the contents of CPU registers, volatile storage, stable storage, and the state of any connected devices. We further break this state down into two categories. We refer to the guest state that is visible to the guest as the *guest-visible state* and the guest state that is visible to the VMM as the *VMM-visible state*. These may differ for two primary reasons. First, there may be state that the guest simply cannot see, for example, read-only registers or the state of emulated devices. Second, there is some state that the hypervisor knowingly hides from the guest (e. g., to protect the value in a register).

As stated in the chapter introduction, it may be very difficult for a VMI component to act directly on the state. In order to perform meaningful analysis, the VMI component must not just be able to observe the state, but must also be able to make sense of the state. In order to do this, some external knowledge must be applied to construct a useful *view* of the system state. A view is a concise representation of the state that incorporates external knowledge about either the guest software architecture or the guest hardware architecture. More specifically, a view may be zero or more of the following:

1. The view may be a portion of the state.
2. The view may be a permutation of the state.
3. The view may be an interpretation of the state.

In some cases, one might be interested in focusing on a particular portion of the state. In this case, external knowledge must be applied to identify and locate the portion of the state that one is interested in. For example, extracting just the process list from the entire state requires an understanding of how the guest OS stores this information and in what data structures.

Permutation of the state may be necessary to identify (dis)similarity between states. If we continue with our example of the process list, we might consider two process lists equivalent if they contain the same processes even if they occur in a different order. This, again, requires external knowledge to be applied. In order to permute the items in the data structure containing the process information, this data structure must be well understood.

Finally, the view may be an interpretation of the state. Defining exactly how this interpretation is performed is situational, but may be as simple as counting instances of a data structure. To further the on-going example, the processes in the process list are counted and the sum is saved while the process list itself is discarded. That is, the number of processes currently running is information that has been interpreted from the process list. As in the above examples, this, again, requires knowledge of the appropriate data structures to complete.

3.2.2 View-Generation and Aggregation

A view is the "essence" of a system state with respect to one's particular application. In order for this "essence" to be extracted, the semantics of every single represented bit must be well-known. When we say that the semantics of a bit is known, we mean that its meaning within the context of the guest system is known. Once this "essence" has been extracted, we may work with it and the challenge set forth for VMI in the chapter introduction has been addressed. Hence, the process of generating a view is vital in any VMI application and we may benefit from a closer look at view generation.

In order for the meaning of every bit within the view to be known, the process in which a view is generated must incorporate knowledge of *the virtual hardware architecture* and/or *the guest system software architecture* (i. e., the combined architecture of the guest OS and all software running on the guest OS). Making use of knowledge of the virtual hardware architecture allows us to translate guest virtual addresses to guest physical addresses, for example, as this is a process that is define by the hardware architecture. On the other hand, making use of knowledge of the guest system software architecture allows us to locate the process list data structure, for example, as this data structure is defined by the software architecture.

In addition to which information is incorporated for view generation, the perspective from which it is generated is equally important. As mentioned in Section 3.2.1, there is guest-visible state and VMM-visible state. In this vein, the view may also be generated either from the perspective of the guest or from the perspective of the VMM. We refer to view generation that occurs from the perspective of the guest as *internal view-generation* as it is happening within the guest and view generation that occurs from the perspective of the VMM as *external view-generation* as it is happening outside of the guest.

While view-generation may be described as generating a view from a state, this is often not enough as one may have to incorporate multiple sequential views to generate a clear picture of what is happening within the VM. This is important for many VMI applications that consider a sequence of views and combine these views into a final aggregated view. For example, a compromised process may sequentially open a large number of sockets. A single view may seem benign as the process only opens a single socket at a time. However, it may become clear that malicious activity is transpiring when looking at a sequence of views. Thus, we introduce the notion of combining several consecutive views of a system run into an aggregated view.

The flow of information is depicted in Figure 3.2. View generation makes use of the observable state along with knowledge of the virtual hardware architecture (HW knowledge) and/or the guest system software architecture (SW knowledge). This information is used to generate views which are then combined to culminate in a final aggregated view. It is important to note that this is a generalization and, depending on the actual application, some components in the information flow may not be considered. For example, an application may rely on *either* internal *or* external view generation, rather than both. Additionally, view generation may rely on knowledge of *either* the hardware architecture *or* the software architecture as opposed to both. Finally, the aggregation step is completely optional as there are some VMI applications that do not require this and can work on a single view.

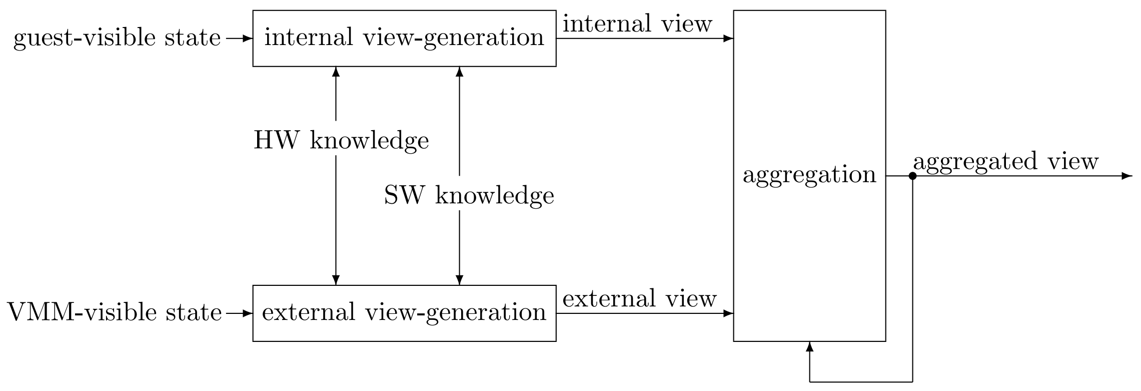


Figure 3.2: Visualization of VMI information flow, where knowledge of the virtual hardware architecture (HW knowledge) and/or the guest system software architecture (SW knowledge) is combined with the guest-visible state and the VMM-visible state in the respective view-generation to create a view. These views may then be combined to create a final aggregated view.

3.3 VMI Patterns

Having taken a look at the general VMI process it becomes clear that view generation is a very important step in any VMI application and that there are various ways to achieve a final aggregated view. That is, VMI applications may perform view generation externally or internally. Additionally, an application may choose to combine internal view generation with external view generation. Furthermore, an application may make use of knowledge of the virtual hardware architecture, the guest software architecture, or a combination of the two. To help in considering various VMI approaches, we present three VMI patterns that can, in combination, describe any VMI approach.

Nance et al. take a similar approach by classifying VMI approaches with regard to *semantic awareness* [54]. In their classification, they consider a system “semantically aware” if it uses knowledge of the guest software architecture and “semantically unaware” otherwise. However, we recognize that systems that are “semantically unaware” according to Nance et al. must still base their knowledge on an understanding of the hardware architecture. To this end, we present three *view-generation patterns* that may be used alone or in any combination as a means for bridging the semantic gap, namely: the *out-of-band delivery* pattern, the *in-band delivery* pattern, and the *derivative* pattern. What follows is an in depth discussion of each pattern and its properties.

3.3.1 Properties

We begin by presenting the various properties that each pattern may have.

3.3.1.1 Guest OS Portability

Guest OS portability refers to a property that allows the same VMI mechanism to work for various guest OSs without major changes. In order to achieve guest OS portability, the underlying VMI mechanism may not rely on knowledge of the guest OS itself, but rather on knowledge of the virtual hardware specifications. We sometimes refer to a component with this property as guest OS agnostic.

For example, Jones et al. make use of the CR3 register in order to track processes [38]. How this register is to be used within the memory management unit (MMU) is specified by the x86 architecture and all OSs running on this hardware and using the MMU must conform to these specifications. Thus, this basic method can be used to track processes in various guest OSs without change as long as the OSs support virtual memory.

3.3.1.2 Binding

Litty et al. introduced the notion of non-binding information [46], this being information that is not bound to the assumptions it relies on. This is often a concern in monitoring any software system. Consider a kernel data structure that must be monitored and the information we have is that this data structure can be found at a particular offset with a specified size. The problem that we run into is that the information about the position and size of the data structure is not bound to any assumption that we can rely on. That is, if the data structure is moved (for malicious or benign purposes) and we are not made aware, we are basing our monitoring on information that is no longer correct. Based on this, we consider an approach to have the binding property if it relies on information that is bound to assumptions that cannot change during run-time.

3.3.1.3 Isolation

Isolation is a straightforward property that states that an introspecting component cannot be influenced in an unintended manner by component within the guest OS. This is standard property that we generally take for granted when working in virtualized systems, however we must be aware that the semantic information and view generation may not always be completely isolated from the guest OS.

3.3.1.4 Inspection of Suspended VM

This is another fairly straightforward property and it describes the ability to inspect a VM while the VM itself is inert. This may be important for static analysis of a VM, as is the case in digital forensics. In such a case it may be beneficial to inspect a suspended VM as this gives the benefit of inspecting a machine whose volatile storage is intact without having the concern that the system may cause further damage.

3.3.1.5 Full State Applicability

This is a very interesting property that describes the coverage of an approach in terms of the state. That is, there are some approaches that can be used to inspect and interpret the *entire* state of the VM. This property describes such approaches. Conversely, there exist approaches for which some parts of the state cannot be inspected and/or interpreted. A very simple in-guest example of this is a host-based monitor that runs in user space. Such a monitor is unable to monitor the memory of the kernel or of other processes due to inaccessibility and is therefore not full state applicable.

3.3.2 Out-of-Band Delivery

This pattern is depicted in Figure 3.3 and is the most commonly used method for bridging the semantic gap. Here semantic knowledge is *delivered* to the external view-generating component in an *out-of-band* manner. That is, the external view-generating component receives the semantic knowledge in advance, before VMI begins. For example, the VMM may make use of a previously delivered symbol table based on the guest OS kernel to determine the position of key data structures.

The view-generating component makes sole use of knowledge of the software architecture. Since this knowledge is delivered in an out-of-band manner, this component is only fit to create a view based on the knowledge it currently has. If a malicious entity changes the software architecture (e. g., by inserting some hidden data structure), clearly the view-generating component is no longer fit to provide an accurate view. The semantic knowledge (i. e., the software architecture description) that the view-generating component uses is in no way bound to the actual software architecture of the running guest. For this reason, such an approach is non-binding.

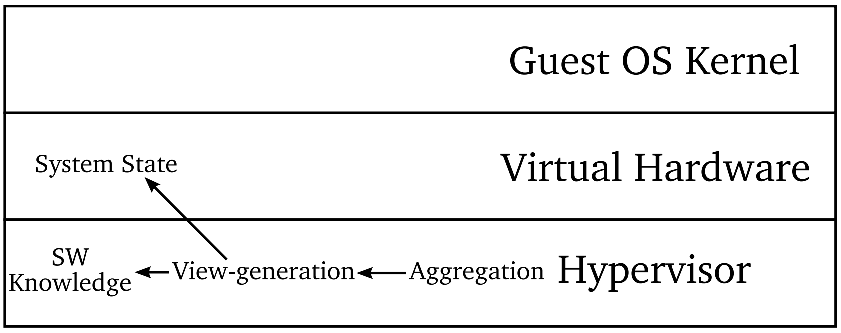


Figure 3.3: *Out-of-Band Delivery:* The view generation makes use of knowledge of the software architecture, which was delivered out-of-band, to interpret the system state from within the hypervisor.

Very closely related to the non-binding nature of this pattern is the lack of guest portability. That is, if the guest system software is changed or replaced completely, new knowledge must be delivered to the view-generating component or in the worst case, the view-generating component must be constructed anew to accommodate the new system software.

As seen in Figure 3.3, this pattern makes use of external view generation. The fact that the view generation is performed externally results in two properties of this pattern. First, the component is isolated from the guest. Therefore, any malicious entities which may have compromised the guest OS cannot influence or corrupt view generation. Second, the VM does not need to be running while the view generation takes place. This allows one to mitigate the external effects of keeping a compromised system running. For example, it is beneficial to suspend a system which acts hostile to neighbors on a production network while still being able to examine it.

This pattern is easily applied and implemented when compared to the derivation pattern, which is detailed below. In a practical example, monitoring any kernel-level data structure externally requires delivering the address and the layout of the data structure to the monitoring component. This information can generally be obtained through debugging methods.

3.3.3 In-Band Delivery

The in-band delivery pattern is depicted in Figure 3.4 and it describes an approach in which an internal component creates a view and delivers this view to the VMM. Since the view-generating component is internal, it may make use of the guest OS's inherent knowledge of the software architecture. That is, semantic knowledge is being *delivered* in an *in-band* manner.

Upon closer inspection, this pattern does not so much bridge the semantic gap, but

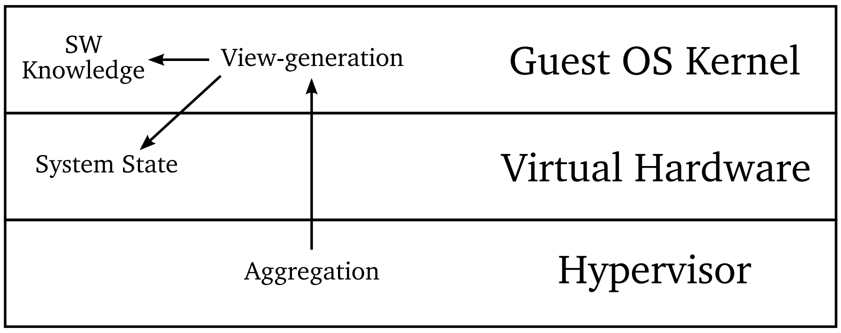


Figure 3.4: *In-Band Delivery:* The view generation makes use of its inherent knowledge of the software architecture to interpret the system state from within the guest OS and delivers the view to the hypervisor in-band.

rather avoids it. In fact, one could rightfully argue that this is not a method for bridging the semantic gap in the purest sense since the view is generated from within the monitored guest OS. However, we include this pattern for the sake of completeness and because it can be successfully combined with other patterns.

Having an internal view-generating component results in a few disadvantages. First, since view generation takes place within the monitored guest OS, the component that implements view generation is susceptible to compromise from any malicious entity that has compromised the monitored guest OS. In essence, it is necessary to trust a component which may be compromised. Second, an internal component does not support the ability to examine a suspended system, creating possible inconsistencies and resulting in the inability to mitigate external effects. Third, an internal component will always interfere with the system being monitored and finally, there may be parts of the state not visible to the view-generating component. Additionally, as with the out-of-band delivery pattern, this pattern is neither binding nor guest OS portable.

We present this pattern as only making use of guest software knowledge. While it is practically possible to create a view-generating component that makes use of hardware knowledge from within the monitored guest OS, such view generation is *always* best done externally for one primary reason: Any view-generating component that makes use of hardware architectural knowledge can be implemented from within the VMM and gains all of the benefits associated with such an approach (e. g., isolation, consistency, etc.). On the other hand, implementing a view-generating component in the guest OS brings no advantages over an approach in which the component is implemented externally. For this reason, we argue that while implementing such a component internally is practically possible, there is absolutely no added value and is therefore not discussed further.

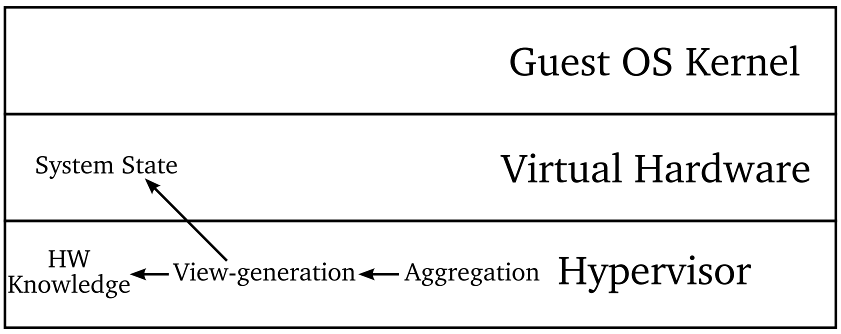


Figure 3.5: *Derivation:* The view generation makes use of knowledge of the hardware architecture to interpret the system state from within the hypervisor.

3.3.4 Derivation

The final VMI pattern is depicted in Figure 3.5 and it moves away from the more common approaches. This approach has the VMM *derive* information through semantic knowledge of the hardware architecture. For example, understanding a particular architecture and monitoring control registers within a CPU provides us with some semantic knowledge.

This pattern makes use of the external view-generating component, thus leading to an approach that is isolated from the guest OS as well applicable in the inspection of a suspended VM.

Further, it can be seen that the view-generating component constructs its output based on knowledge of the hardware architecture alone. This results in two additional advantages. First, this pattern is completely guest portable as the view-generating component has no reliance on the guest software architecture. Second, this is a binding approach in contrast to the two delivery methods. That is, the hardware architecture description is bound to the hardware architecture of the virtual machine because there is no way for a malicious entity that compromised the guest OS to change the virtual hardware architecture. A malicious entity has no way of changing the hardware, and whatever changes it may apply to the software will not effect the view-generating component's ability to create a view.

This seems like the ideal pattern, though in practice it becomes difficult to implement as it requires extensive knowledge of the hardware. In addition, there is some state for which hardware knowledge is not helpful in view generation. This is due to the fact that there is information that cannot be extracted by only making use of hardware architectural knowledge, making this approach not full state applicable. For example, the arrangement of a data structure that performs some bookkeeping for a process is impossible to derive based on hardware knowledge if this bookkeeping is completely independent of the hardware. To what extent this limitation exists is explored further in Chapter 6.

3.4 Combination of Patterns

The patterns that have been discussed up to this point are in their purest form. It is, of course, possible to combine such patterns. In fact, it is sometimes beneficial to use a drawback of one pattern in combination with another pattern to one's advantage: A component of rootkit detection often involves trying to determine whether two mechanisms, one often at a higher level than the other, report the same status about some OS data structure. Such an approach is possible by combining the in-band and out-of-band delivery patterns and relying on the fact that a capable malicious entity will compromise the internal view-generating component or the guest OS itself, thus leading to an inconsistency between the two views created, which is then detected.

Property	*Delivery*		*Derivation*
	in-band	*out-of-band*	
Guest OS portability	—	—	✓
Binding	—	—	✓
Isolation from guest OS	—	✓	✓
Inspection of suspended VM	—	✓	✓
Full state applicability	—	✓*	✓**

Table 3.1: Comparison of the view-generation patterns (*only when combined with a derivative approach; **only when combined with an out-of-band delivery approach)

We have summarized the properties of the different patterns in Table 3.1. When combining such patterns, it is important to carefully consider how the properties of these patterns will interact with each other. For instance, *full state applicability* can only be achieved in combination. That is, no single approach is capable of interpreting all state. For example, an out-of-band delivery approach may need to calculate a guest physical address from a guest virtual address. In order to complete this, knowledge of the hardware architecture is required and no knowledge of the guest software architecture is of help. Specifically, one must know how the translation from a guest virtual address to a guest physical address takes place and what data structures are involved. One might argue that such information might be delivered along with knowledge of the guest software architecture as the software might need to implement such functions as well. While this may be true, this knowledge still ultimately comes from knowledge of the hardware architecture. That is, while virtual to physical address translation might be implemented in the kernel, this is has been implemented based on knowledge of the hardware architecture. Conversely, a derivative approach alone cannot interpret all state. For example, knowledge of the hardware architecture will not help in locating the process list within the guest kernel as this data structure is completely independent of the hardware architecture.

On the other hand, deciding what approach is best for a given application is not as simple as combining all patterns in order to get the benefits of each. An advantage of one pattern will not necessarily negate the disadvantage of another. For example, a combination of a delivery pattern with the derivation pattern will not result in an approach that is guest portable simply because one of the patterns has such a property. In fact, in this case, the opposite is true: The disadvantage of one pattern negates the advantage of another.

There are additional cases in which the properties are independent, for example, a combination of a delivery and a derivation pattern, though this time with respect to the binding property. A combination of two such patterns can neither be classified as binding nor as non-binding. There is simply some information within the system that is the result of bound knowledge and some information within the system that is the result of unbound knowledge. How this information is to be used needs to be carefully considered.

Finally, some patterns are simply not appropriate for a given application. For example, using the derivation method alone to monitor changes in a file on a disk is a poor choice as it is impossible to derive the exact sector of a disk on which the appropriate portion of a file lies without knowledge of the file system structure in use.

3.5 Summary

In this chapter, we have taken a closer look at VMI and its application. We began by introducing a definition that encompasses all VMI techniques based on the foundation that Garfinkel and Rosenblum laid [26]. We then introduced the single challenge that all VMI applications must face, namely the semantic gap [14]. This describes the challenge that one faces when trying to interpret the binary state that one sees from the perspective of the hypervisor.

To consider this challenge more closely, we explored a general VMI process. That is, we considered the steps necessary to interpret the raw state and construct a view of the guest VM's state. We identified that view generation is synonymous with bridging the semantic gap. We then considered the various methods for creating a view with regard to the information that is included and the perspective from which the view is generated.

Having looked at the process, we identified three patterns with which one can classify all possible view-generating approaches, namely in-band and out-of-band delivery, and derivation. Furthermore, we presented a thorough discussion of each pattern's properties as well as their advantages and disadvantages for view generation. The patterns themselves allow us to consider various approaches and provide a taxonomy for VMI applications that is helpful in discussion. We make use of this in our own work when considering other approaches and argue that having some form of taxonomy in any field makes considering new approaches more methodical.

As we have found our taxonomy of VMI methods an invaluable tool for discussion and analysis of VMI methods, we urge readers to consider approaches based on this taxonomy. As the taxonomy is based on the single challenge that all VMI methods share, this is helpful and logical manner to consider various approaches. The patterns are able to represent all VMI approaches in a consistent way, making them comparable and allowing one to reason about their properties and appropriateness for a given application.

Chapter 4

Related Work

Having introduced the various view-generation patterns, we demonstrate their usefulness and put them to work by presenting related work with respect to these patterns.

4.1 Delivery Approaches

Since the work done by Garfinkel and Rosenblum [26], there has been quite a bit of work taking similar delivery approaches, all of which can be described with the patterns we identified. The VIX tool suite [32] is an implementation of the out-of-band delivery pattern for forensic analysis. This system uses delivered knowledge about the guest OS in order to implement common Unix tools such as `ps` and `lsmod`. Systems such as HookSafe [86], NICKLE [66], and Rhee et al.'s system [62] strive to monitor or protect the integrity of kernel code or data structures. These systems all take a delivery approach, whereby NICKLE needs minimal delivered information as it works by shadowing the entire kernel memory space. This means that NICKLE is not interested in specific data structures, but rather the offset of the kernel memory space as a whole. Additionally, XenAccess [56] uses a delivery approach to provide a library for general VMI functionality such as memory and disk access. Due to the flexible nature of XenAccess, Dolan-Gavitt et al. were able to run common forensic tools on a VM though its use [19]. Finally, Insight also takes an out-of-band delivery approach, but in addition to common delivered knowledge, Insight is able to parse the source code of the target kernel [69, 70]. This allows Insight to perform an *used-as analysis* to allow one to identify the data type of any data structure in memory.

Both Lares [57] and Xenprobes [61] take a combined in-band and out-of-band delivery approach to VMI by placing hooks into the guest OS from the hypervisor. This allows these systems to partially use semantic information that is inherent to the guest OS. VMwatcher[36] takes a similar approach, but it combines an out-of-band delivery approach with an in-band delivery approach to perform cross-view validation. That is, the system compares the process list as collected from the hypervisor with the process list as collected from within the guest OS to look for inconsistencies. IntroVirt [39] also makes use of an in-band delivery approach to bridge the semantic gap. This system defines “predicates” that are executed to detect when vulnerabilities are exploited. These predicates may make use of functions within the guest OS.

We mention in Chapter 3 that in-band delivery approaches suffer due to the fact that they run within a possibly compromised guest OS. While this is true, there has been work to mitigate the risks associated with this. This is an interesting approach as bridging the semantic gap becomes straightforward due to the fact that code running within the guest OS may benefit from the semantic knowledge that the running system has. Such approaches try to implant code within the guest OS, however they take extra care to secure this code from the hypervisor. Sharif et al. introduce the notion of Secure In-VM Monitoring (SIM) [73]. This approach attempts to load code into the VM and protects this code by using gates that protect access to the SIM code through clever use of shadow page tables. Gu et al. propose a similar approach although they make use of the guest OS’s protection mechanisms to implant a process and have it seem as if it is a process already running within the guest [31]. This allows them to run a process in the context of the guest OS with a “degree of stealthiness”.

Finally, there exist two approaches that take a delivery approach, but are very clever in how this semantic knowledge is gathered. Rather than using traditional debugging symbols or symbol tables, these systems make use of an OS identical (or at least similar) to the guest OS to learn or extract semantic information about the guest OS. The first system, introduced by Srinivasan et al., moves a process out of the guest OS into a security VM, but passes system calls back the guest OS to be executed there [76]. This allows a security critical processes to run in isolation while accessing the kernel of the guest OS through system calls. The second such system, Virtuoso, performs a simple task within an OS and records the instruction trace [18]. From these instruction traces Virtuoso is able to automatically generate a code module that can be run from the hypervisor and performs the same job as the initial task. In both of these systems there is no semantic knowledge that is ever explicitly delivered. However they make use of OSs, which must be similar enough to the guest OS, to bridge the semantic gap. This is still delivered knowledge, however the delivered knowledge takes the form of an entire OS. In the case of Srinivasan et al.’s system, the security VM provides a context for the process to run within. That is, the OS running within the security VM must be similar enough to the guest OS as to host the process. Additionally, the system call mechanism must be identical to that of the guest OS for the forwarding of the system calls to be possible. On the other hand, Virtuoso takes an approach in which it makes use of a similar OS beforehand to create a module which will run in a similar system later. Here, again, the OS provides the context for the initial task to run. This is then recorded and replayed from a different perspective.

4.2 Derivative Approaches

Litty et al. present a system they call Manitou [47]. It uses the paging mechanisms of x86 processors from Intel and AMD to perform integrity checks on code pages before execution. They further this work to create a system called Patagonix for reporting running processes and maintaining the integrity of code pages [46]. This work uses knowledge of the hardware architecture, specifically the paging mechanism, to perform its intended tasks. Jones et al. [37, 38] take a slightly different approach, but also present a system that implements the derivation pattern. Their system makes use of the paging mechanism and the memory management unit (MMU) in x86 and SPARC hardware architectures to identify running processes.

The work of Vogl et al. [82] is another interesting example of how a derivative method can be applied. This work makes use of debugging and performance monitoring mechanisms within the x86 hardware to perform instruction tracing as well as implementing a shadow stack to protect against stack-based shellcode attacks. The authors make use of performance monitoring counters (PMCs) in conjunction with the last branch record (LBR) to collect information about branching statements that have been executed, then recreate the instruction trace without having to trap each individual instruction. In addition, they make use of the LBR to implement a shadow stack. That is, they keep a copy of the stack in the hypervisor and on a return statement they can verify that the return address has not been modified.

A final approach that combines a delivery approach with a derivative approach for system call tracing is the Ether system [17]. This system makes use of paging mechanism in x86 processors to trap system calls. There have been several systems that make use of the same approach for system call trapping [78, 77, 53, 20, 91], though none take a purely derivative approach and rely on some delivered knowledge.

4.3 Looking Forward

It becomes evident that there has been quite a bit of research done in the direction of delivery approaches and relatively little in the direction of derivative approaches based on the related work. The presented derivative approaches show that such an approach is feasible, but that there is still much potential. Developing new derivation methods takes a great deal of understanding of hardware architectures and there simply are limitations to such an approach. One cannot derive the meaning of every bit of the state based solely on knowledge of the hardware architecture. However, to what extent such derivative approaches can be useful is yet to be explored fully. It is exactly this potential that we explore in this thesis.

Chapter 5

Feasible Goals

Virtual machine introspection is often used for security applications. In many proposed VMI systems, the systems' creators strive for transparent VMI mechanisms [8, 17, 30, 53, 62, 66]. That is, they attempt to argue that the presence of introspection features are undetectable from within the VM.

We take the position that achieving true VMI transparency is infeasible and, in some cases, ill-advised. We support this with the thesis of Garfinkel et al., that "building a transparent VMM is fundamentally infeasible, as well as impractical from a performance and engineering standpoint" [24]. Garfinkel et al. go on to argue that, from an observation standpoint, there is little to be concerned about. As virtualization becomes ubiquitous, simply detecting the presence of a VM will no longer be indicative of monitoring taking place and that due to this, lack of transparency need not be a concern.

While we agree that as virtualization becomes ubiquitous, running inside of a VM is no longer an indicator that introspection is taking place, we argue that transparency remains a concern with respect to VMI. The issue of transparency has simply been relocated. That is, as virtualization becomes ubiquitous, malicious entities will turn their attention from detecting virtualization to detecting VMI mechanisms.

This is a very important distinction. We refer to the property that states that virtual hardware is indistinguishable from physical hardware as *VMM transparency* and the property that states that virtual hardware that does not perform any introspection is indistinguishable from virtual hardware that does perform introspection as *VMI transparency.* In Section 5.1, we will argue that the thesis presented by Garfinkel et al. for VMM transparency can be extended to VMI transparency as they can be reduced to the same problem.

In place of transparency, we suggest two feasible properties that better describe the level of protection against attacks, namely *evasion-resistance* and *evasion-evidence.* Both properties can be achieved by a construction that we call hardware-rooting.

5.1 Transparency

Garfinkel et al. concluded that VMM transparency is infeasible by presenting three types of discrepancies that reveal the virtualized nature of a system [24].

Logical Discrepancies. The first type is logical discrepancies between the virtual hardware and the physical hardware. This might take the form of instructions that have varying side-effects depending on the (virtual) hardware vendor. Imagine a case in which the architectural specifications do not specify what happens with the input registers after an instruction has been executed. One vendor might leave the input values in the registers, while another vendor might leave some intermediate values in the registers, while yet a third vendor might zero out these registers. This is an example in which a simple hole in the specification leads to discrepancies. One might argue that this can be simply solved by mimicking the target vendor's hardware exactly. While this is an ideal solution, practically it is infeasible on modern hardware architectures due to the size and complexity of the hardware specification. The Intel x86 architecture features over 500 instructions that are compatible with approximately 35 processor families [35].

Resource Discrepancies. The second discrepancy is a result of shared resources. As the hypervisor must share resources with the guest, the fact that sharing takes place might become evident through resources that are unavailable or left in a state which might be unexpected. Garfinkel et al. use the example of the TBL. The TLB is a cache used by the hardware to accelerate virtual to physical address translation. As this cache is shared outside of the VM, the TLB may betray the virtualized nature of the environment. One could consider implementing the TLB in software. While feasible, this would bring about a respectable time overhead which would result in a timing discrepancy.

Timing Discrepancies. Finally, the third discrepancy comes in the form of timing discrepancies. This is the result of mechanisms which are not virtualizable and must be implemented in software. This leads to time overheads as well as latency patterns which one might not expect on physical hardware. In the simplest case, the detection of such discrepancies becomes a matter of timing a series of instructions one suspects might be implemented in software. In this case, someone might argue that from within the hypervisor we also have control over the system clock and can conceal these discrepancies by manipulating the system clock. While this is correct, this is only feasible if the system has no access to an external time source. This does not have to be an overt time source, such as an NTP server, but any external contact can be used to, even implicitly, convey time. Playing warden to all possible communication that might include timing information, implicitly or explicitly, is infeasible.

These arguments can also be extended to argue that VMI transparency is infeasible. Logical and timing discrepancies may come in the form of instructions that need to be emulated for one's VMI component to function. Resource discrepancies may come as a result of virtual devices or resources that a VMI component needs to tap into, such as a network or storage device. Finally, the most common type of discrepancy in a VMI component is in the form of timing discrepancies. By the definition of VMI presented in Chapter 3, a VMI component will either collect information or manipulate the VM. Either of these actions will incur a time overhead beyond what is normal for the operation of the system and the resulting timing discrepancy will be detectable.

In fact, we can generalize the arguments of Garfinkel et al. further. They conclude that it is infeasible to make hardware HW indistinguishable from virtual hardware vHW due to these discrepancies. We argue that it is infeasible to make virtual hardware vHW indistinguishable from virtual hardware vHW'. We can quickly see that both of these statements can be reduced to the problem statement, "is hardware platform a distinguishable from hardware platform b?". This is independent of whether a or b are virtual or not. We may consider that both a and b are physical hardware platforms from different vendors. If a and b behave differently in any way, they will be distinguishable. As both our claim and that of Garfinkel et al. can be reduced to the same problem statement, we conclude that for the same reasons that Garfinkel et al. found VMM transparency to be "fundamentally infeasible, as well as impractical from a performance and engineering standpoint" we claim that VMI transparency is infeasible. If we assume that vHW' is being exposed by an introspecting VMM, this introspection will either take the form of observable changes in the guest operating system (OS) or extra time spent in the VMM to perform collection or analysis. In either case, this change in behavior will be evident to an adversary that is looking for it.

5.2 Motivations for Transparency

Having argued that both VMM transparency as well as introspection transparency are infeasible, we are left to explore feasible properties to replace transparency. This requires that we consider the motivation for transparency in virtualization-based security applications. We identify exactly two such motivations.

5.2.1 Avoiding Observation

A VMI system that constitutes a forensic or sandboxing tool for software analysis would benefit from transparency. Malware authors often try to hinder the analysis of their malware. One way to hinder a dynamic analysis is to build versatile detection routines for well-known sandboxing and analysis environments into their malware. If this malware detects it is running in an analysis environment, it may take action to hinder analysis. If a VMI system could be designed to be truly transparent, such detections would not be possible, hence all malware could be observed.

Remaining transparent while observing an entity mitigates the observer effect. That is, one can only fully observe the natural behavior of an entity if that entity is unaware that it is being observed. It is often the case that modern malware will behave differently when running in a sandbox, for example. If it were possible to observe this malware in a truly transparent manner, one could always be sure they are observing the natural behavior of that malware.

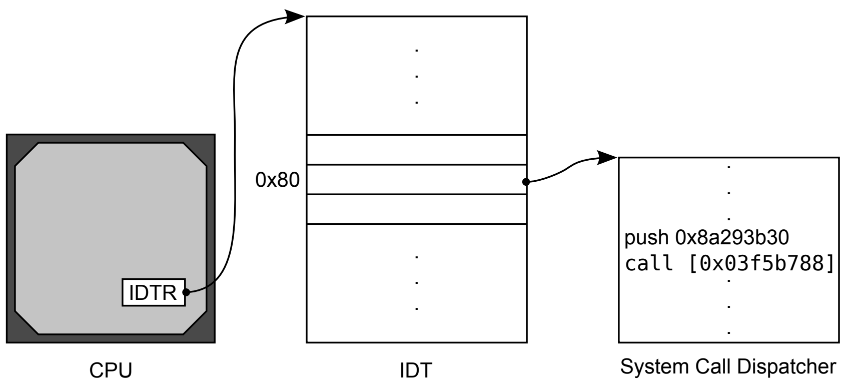
Unfortunately, by the nature of the observer effect it is unavoidable. That is, even if the influence of observation is marginal, the influence remains and an entity willing to put forth the effort can and will detect this influence. In the case of a virtualization-based mechanism this influence may be in the form of altered guest OS state, unexpected results to specific actions, or extra time spent in the VMM for collection or analysis purposes. One can take steps to mitigate the observer effect and, in fact, a virtualization-based approach is already a great step in mitigating this effect over an in-OS solution, however malware observation and analysis will remain a game of back and forth between malware authors and security experts.

5.2.2 Protection Against Evasion

A VMI system often implements a mechanism to enforce certain security policies. A naive approach to protecting this mechanism from evasion attempts might be to hide its existence from an entity within the VM. The logic in this case is that if a malicious entity is unaware of the presence of a mechanism it becomes difficult to evade. However, we make the distinction that a system that cannot be evaded does *not* follow from a transparent system, even though it is often a motivator. With a priori knowledge of a system, even a transparent system can be evaded and a system which is not transparent can be made resistant against evasion attempts. This is an important distinction. Although these properties are often discussed together, a transparent system and a system that cannot be evaded are two separate goals.

5.3 Feasible Goals

Based on the realization that transparency and evasion resistance are two independent goals, protecting a virtualization-based security mechanism from evasion attempts is feasible without transparency. In order to describe these properties, we draw an analogy between protecting software from evasion and protecting hardware from tampering by introducing two properties, namely *evasion-resistant* and *evasion-evident*. These properties are analogous to the terms “tamper-resistant” and “tamper-evident” when protecting hardware. That is, evasion-resistance refers to the property that pro-actively hinders the evasion of a security component, while evasion-evidence refers to the property that reactively makes evasions evident such that they can be identified later.



```
push 0x8a293b30
call [0x03f5b788]
```

Figure 5.1: The system call dispatcher is rooted in the IDTR through the IDT.

5.3.1 Hardware Rooting

In order to define the requirements for evasion-resistant and evasion-evident components, we begin by defining what is meant by *hardware rooting*. We must begin with a feature of the hardware (e. g., a register) whose purpose is strictly defined by the hardware architecture specification to contain the location of one or more data structures. We will refer to this hardware feature as the *hardware anchor*. If we can start from such a hardware anchor and follow references to further static data structures or code segments, thus building a chain to a critical data structure, we call this critical data structure *rooted in hardware*.

Any manipulation of a data structure or code segment along the reference chain from the hardware anchor to the protected data structure is easily detectable by storing the initial state of the data structures or code segments along the chain in the hypervisor, then verifying their integrity during subsequent traversals of the reference chain. For this reason, it is also important that the data structures and code segments along the reference chain remain static during the normal operation of the guest or that one know beforehand all legal data structures or code segments that a pointer may reference. In this manner any manipulation along the chain will not go unnoticed.

A simple example of hardware rooting is depicted in Figure 5.1. This figure shows how the system call dispatcher is rooted in a hardware anchor (i. e., the IDTR). The x86 hardware architecture specifies that the Interrupt Descriptor Table Register (IDTR) must hold the location and size of the Interrupt Descriptor Table (IDT) [35, 5]. In addition, the x86 hardware architecture specifies the layout of the IDT. The software must comply with these specifications in order for the system to function properly.

Locating a hardware anchor requires an in-depth understanding of the hardware architecture in question and is often a process of manually going through the architectural specifications to find hardware features (generally registers) that reference specific data structures or code segments. Admittedly there will be relatively few such hardware anchors, however finding a way to root a data structure in hardware has benefits in the

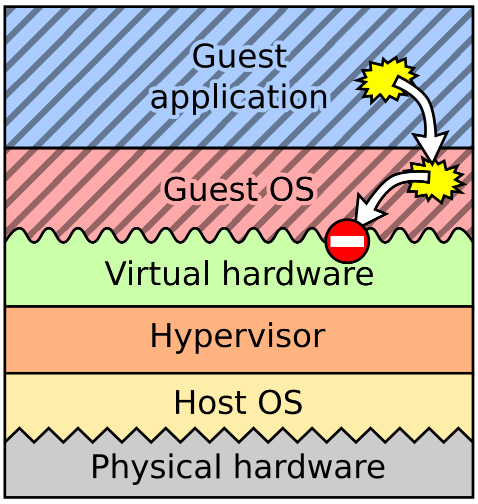


Figure 5.2: Any entity within the guest OS is confined to VM and cannot manipulate the system at a level lower than the virtual hardware.

form of the properties discussed below.

5.3.2 Evasion-evidence

An evasion-evident mechanism is a mechanism which does not explicitly prevent evasion, but is constructed in such a manner that any attempt at evasion can be detected after the fact. More precisely, it is impossible for an attacker to evade the *correctly implemented* mechanism in an *ideal system* without these changes going unnoticed to the mechanism. We define a correctly implemented mechanism as a mechanism that perfectly enforces the policy that it was designed to enforce with no flaws or errors. In the same manner, we define an ideal system as a system that perfectly implements its design and contains no flaws or errors. We introduce the terms *correctly implemented* and *ideal system* simply to highlight the fact the we realize there may be a way to circumvent any mechanism if there is a bug in the system. However, assuming that everything is implemented perfectly, there is no way for a malicious entity to evade the mechanism and go unnoticed.

In order for a mechanism to be evasion-evident, the protected portions of the VM state must be rooted in hardware and the state of each data structure or code segment must be . If we consider Figure 5.1, a mechanism monitoring the system call dispatcher is evasion-evident as any change to the system call dispatcher is observable from the hypervisor and relocating it would require making changes to the IDT. In turn, any change to the IDT is observable from the hypervisor and relocating it would require making changes to the IDTR. Finally, the IDTR is our hardware anchor and it is also observable from the hypervisor. This results in the possibility that an alteration may take place, however the change will be evident as the reference chain can be followed starting from the hardware anchor (i.e., the IDTR). Finally, as depicted in Figure 5.2

any entity (malicious or otherwise) is unable to move to level lower than the virtual hardware.

5.3.3 Evasion-resistance

An evasion-resistant mechanism is a mechanism which is impossible for an attacker to circumvent when *correctly implemented* and deployed in an *ideal system.* Here we define both a correctly implemented mechanism as well as an ideal system in the same manner that we did in Section 5.3.2.

There are two requirements in order for a VMI mechanism to be evasion-resistant. First, the monitored or protected portions of the VM's state must be rooted in the virtual hardware as discussed in Section 5.3.1. Second, each involved piece of VM state along the relevant reference chain must be protected such that it cannot be manipulated in violation of policy. If both of these requirements are met the mechanism is evasion-resistant.

If we continue with our example using Figure 5.1, the mechanism monitoring the system call dispatcher is evasion-resistant if, in addition to being rooted in hardware, the system call dispatcher and the IDT can be protected against manipulation. This might be accomplished by making use of memory protection mechanisms such as paging or segmentation. This results in a situation where each link in the reference chain can neither be manipulated nor relocated and thus, the mechanism is evasion-resistant.

5.4 Mitigating the Observer Effect

Having shown feasible goals for VMI to replace transparency when the goal is protection against evasion, we would like to address the goal of observation. As stated in Section 5.2, avoiding observation remains a large driver for transparency. There are many applications, such as malware analysis or live forensics, for which true transparency would be of great benefit. However, as we presented, transparency is an infeasible goal.

This is, however, not to say that VMI is not a promising tool for such applications. In fact, VMI raises the bar toward transparency substantially, especially in cases where hardware-assisted system virtualization is leveraged. We simply must be aware that the observer effect is always, by its nature, present. That is, for any VMI application one must be aware that full transparency is not feasible, but rather work to mitigate the observer effect as much as possible.

In contrast to solutions that make use of emulation or in-system hooks, hardware-assisted system virtualization makes it very difficult to determine that monitoring is taking place. A dedicated adversary will always be able to detect the presence of a monitoring component, but hardware-assisted system virtualization makes this more difficult in comparison to traditional sandboxes [90, 10, 2, 25, 4].

5.5 Summary

Transparency remains an important aspect in security applications, however it simply cannot be mistaken as a concrete goal due to the fact that both VMM transparency and introspection transparency are infeasible from a practical standpoint. However, having considered the motivations behind the need for transparency, we have identified that VMI-based tools mitigate the observer effect when compared to in-OS solutions. Exploring these motivations further led us to the conclusion that in some circumstances the need for transparency is not necessary and to this end, we introduce evasion-resistance and evasion-evidence.

Chapter 6

Derivative Methods

In this chapter, we investigate the potential of Intel's x86 architecture (i. e., Intel IA-32 and IA-32e) and the virtualization extensions from both Intel (VT-x) and AMD (SVM) for derivative view-generation. We identify and describe methods for extracting security relevant information about a guest system from its low-level state through knowledge of the hardware. These methods allow one to enumerate the running processes, monitor system calls, or track network connections on a per-process basis in a completely guest OS agnostic manner.

6.1 Threats

There are certain threats to system security which can be countered by mechanisms that would benefit from a derivative approach. These threats are presented here. The building blocks for building derivative VMI methods are then presented in Section 6.2.

6.1.1 Interrupt Hooks

Setting interrupt hooks is a common tactic among rootkit authors, especially hooks in the system call mechanism. This allows the rootkit to be alerted to certain activities within the kernel and to intercept the communication between a process and the kernel. To further understand this threat, it is important to understand the interrupt mechanism as it is implemented in the x86 architecture.

The standard interrupt mechanism makes use of a system register, the Interrupt Descriptor Table Register (IDTR). This system register contains the address and size of the Interrupt Descriptor Table (IDT), which, in turn, holds the addresses of the various interrupt handlers. When an interrupt occurs, the system refers to the IDT to determine the address of the appropriate handler, then invokes this handler. These interrupts are generally divided into two categories: system interrupts and user-defined interrupts. System interrupts make up the first 32 entries of the IDT and include general protection

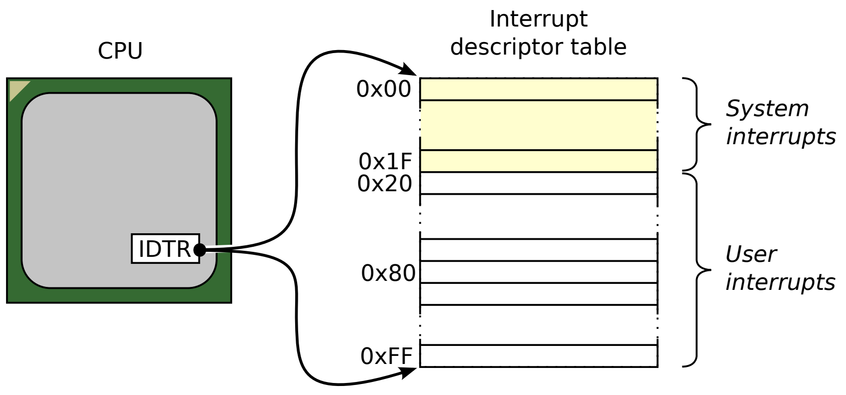


Figure 6.1: The Interrupt Descriptor Table Register (IDTR) holds the location and size of the Interrupt Descriptor Table (IDT), which is responsible for storing the locations of the interrupt handlers.

faults, page faults, division by zero, etc. User-defined interrupts may reside in the table anywhere after the first 32 entries. This is depicted in Figure 6.1.

Considering the interrupt mechanism, there are three possible stages at which an attacker could possibly perform interrupt hooking:

1. Manipulate the IDTR to point to a malicious IDT
2. Manipulate the IDT entries to point to malicious handlers
3. Manipulate the handlers to perform malicious activities

In order to protect against interrupt hooking, verifying the integrity of the interrupt mechanism at *all* possible stages is imperative.

6.1.1.1 System Call Hooks

A system call is a software interrupt, though the mechanism is a bit more complex as there are two primary ways a system may choose to handle them. The first is through the `int` instruction which triggers a user-defined software interrupt as discussed in Section 6.1.1. The second method is referred to as the fast system call method and it uses dedicated `SYSENTER` or `SYSCALL` instructions. Further technical differences are irrelevant in this section, but note that this mechanism makes use of machine specific registers (MSRs) that point to the system call dispatcher (i. e., an interrupt handler) directly. Modern operating systems generally support both mechanisms.

Another critical component of the system call mechanism is the system call table. This is generally a kernel data structure that contains the entry point addresses for the individual system call handlers. That is, the system call dispatcher consults the system

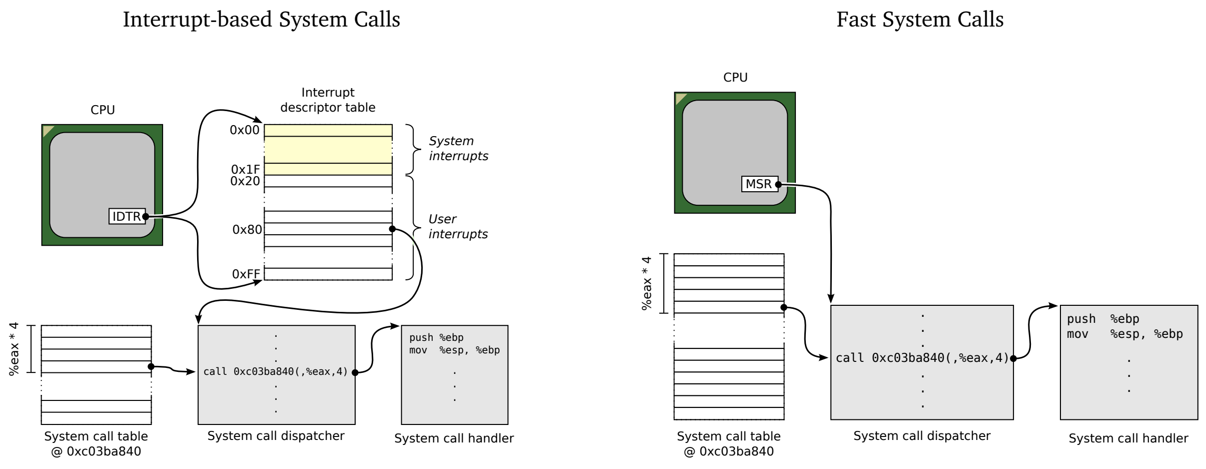


Figure 6.2: The system call mechanisms in the x86 architecture.

call table to determine the memory address of the respective handler, then invokes that handler. This relationship is depicted in Figure 6.2 for both system call methods.

Considering these two methods for system call invocation, there are five possible stages at which an attacker could possibly perform system call hooking:

1. Manipulate the IDTR to point to a malicious IDT
2. Manipulate the IDT entries or MSR values to point to malicious dispatchers
3. Manipulate the dispatchers to perform malicious activities
4. Manipulate the system call table entries to point to malicious handlers
5. Manipulate the handlers to perform malicious activities

6.1.2 Information Hiding

Hiding information from the guest OS is another common tactic employed by rootkits. Information hiding may include a wide range of goals, but we will focus on three of the most common, namely: process hiding, kernel module or driver hiding, and network activity hiding. All of these may be used by an attacker to disguise his presence on a compromised machine. The issue with such hiding is that, from the perspective of an attacker, there exist several ways to reach each goal. However, there are commonalities which can be exploited by a security application to counter each goal. These are outlined below.

6.1.2.1 Process Hiding

Process hiding is the act of hiding a process or task from tools within the operating system. A competent user or system administrator will expect to see certain processes and an unknown process may cause suspicion. An attacker will want to avoid any such suspicion and therefore employs process hiding techniques.

Process hiding is generally achieved by manipulating the relevant data structures in the kernel. Essentially, one must manipulate the bookkeeping data structures of the kernel such that the the particular process is no longer present in them. Then, when the kernel checks these data structures for a list of running processes, the hidden process is not among them. There is, however, an interesting caveat when performing process hiding. The hidden process must still get scheduled to run. That is, hiding a process so well that the kernel is no longer able to schedule it is counterproductive.

One way to counter process hiding is to perform an enumeration of running processes (tasks) at two different levels. This is referred to as cross-view validation [11]. First, a "low-level" task enumeration is performed. That is, the task enumeration is performed by directly inspecting kernel data structures. Then a "high-level" enumeration is performed by using the OS interface (e. g., using `ps -A` in Linux). The collected information can then be compared and any inconsistencies indicate that process hiding is taking place. For such a technique to work the low-level task enumeration must be performed at a level lower than the level at which the process hiding is taking place.

6.1.2.2 Kernel Module and System Driver Hiding

As with process hiding, hiding kernel modules or system drivers is the act of hiding these kernel-level components from tools within the operating system and thus from the user.

While the goal is similar to that of process hiding, the difference is that these components get loaded into kernel memory and do not possess a separate virtual address space. To complicate things further, these components may continue to perform normally even when the kernel's handle to the particular component has been completely removed. That is, the kernel is completely unaware of such a component and yet it continues to perform its tasks. For example, a Linux module may be loaded into memory, making the code resident in kernel space. A malicious entity may then remove the entry for this module from the module list without unloading the code. This results in malicious code being resident in the kernel without any indication that it is present [16].

Of course, the memory which is in use by a hidden component must be allocated and one could take the approach of monitoring all allocated portions of kernel memory and identifying all unknown components (i. e., components for which the kernel has no handle). However, this approach is highly cumbersome and impractical as it requires semantic information about every piece of kernel information in order to determine if a hidden components exists.

An alternative approach is based on the observation that such a component must get loaded before it becomes hidden and the loading of such components must be handled within the kernel at a specific location, often through a system call. That is, by monitoring the appropriate system call for such components we have the opportunity to recognize when kernel modules or system drivers are loaded and thus alleviate the need for detecting hidden components. We may use the same method described in Section 6.1.2.1, comparing this VMI view to an in-OS view. It is important to note that this method relies heavily on the guest OS. If the guest OS provides methods other than system calls to load kernel code, an approach anchored in a derivative component may not be possible.

6.1.2.3 Network Activity Hiding

Another common practice of rootkits is to hide network activity. That is, an attacker will want to hide all network traffic created by his tools from the OS as this is another common avenue by which users and administrators become suspicious. For example, an attacker may want to set up a back-door that listens on a particular port. The attacker will want to hide the fact that a service is listening as well as any traffic created by this service.

Any traffic created within a VM must travel through the hypervisor as the hypervisor manages all virtual I/O devices that are extended to the VM.[1] This can be taken advantage of and in fact it is impossible for any changes within the guest to hide network activity (or any I/O activity) from the hypervisor. In addition, it is possible to correlate network packets with processes on the system, as we will explain in Section 6.2.1.4.

6.1.2.4 Virtual Machine Rootkits

While not seen in the wild, rootkits that move the currently running OS into a VM have been shown as a proof of concept [68]. Such rootkits make use of the `VMENTRY` instructions provided by the virtualization extensions of modern CPUs. Such rootkits can be thwarted by disallowing the use of these `VMENTRY` instructions from within the guest OS or trapping and regulating their usage.

6.1.3 Network-Based Attacks

Network-based attacks include any form of attack or malicious behavior which is carried out over a network. Ideally these attacks are recognizable when monitoring network traffic.

Traditional Network Intrusion Detection Systems (NIDS) are a common tool in any security solution. These systems listen on a network wire and report traffic that is deemed

[1]There do exist exceptions to this in which devices are passed to the VM at a PCI level, but since these exceptions can be explicitly avoided in cases where VMI is the goal, we disregard them.

suspicious, usually based on a set of signatures, i. e., patterns of malicious network traffic. As mentioned in Section 6.1.2.3, all network activity from a guest OS is visible to the hypervisor, thus introducing a NIDS within the hypervisor is a fairly trivial task. This NIDS can be used to protect the guest as well as to detect misbehaving processes in the guest OS. Furthermore, information from the NIDS can then be combined with contextual information about the process sending or receiving the packet to raise its accuracy.

With this contextual information it is possible to provide connection profiles for individual processes. That is, we can correlate outgoing packets with unique processes, thus creating a network activity profile on a process granularity. These profiles can then be used to detect patterns or match signatures at a per-process level.

6.1.4 Malicious Processes

Malicious process detection is a fairly broad goal and recognizing the subtleties of all malicious processes is unlikely if not impossible through VMI. However, there are some areas in which VMI can be used to support malicious process detection.

Forrest et al. consider detecting malicious processes from a natural immune system point of view [22, 21]. They contend that in order to in order to detect whether a process is acting abnormally, one must be able to model normal behavior. This normal behavior can be modeled in a number of ways. For example, one could consider network activity or connection tracking as suggested in Section 6.1.3.

In addition, one could model this normal behavior with system call traces. Once we have a system call profile for a process, this information can be used to identify misbehaving processes through uncommon activities. Such techniques have been presented using system call traces collected from within the monitored system [22, 34, 21, 42, 64, 92, 44]. By using derivative methods the properties of these approaches could be strengthened.

6.2 Hardware Mechanisms

In this section, we outline some of the hardware mechanisms that become the building blocks of a derivative approach. We look at the x86 architecture from two perspectives. First, we explore the introspective features that the architecture provides. These features include primarily passive information gathering techniques. Second, we show how the virtualization extensions can be exploited for introspection purposes. As we will see, they also allow us to actively intercept certain events and instructions triggered inside the guest.

Information presented in this section is based on specifications found in the Intel and AMD Developer Manuals [35, 5] unless otherwise noted. We reinforced this research through experimentation using the Linux Kernel Virtual Machine (KVM) [40].

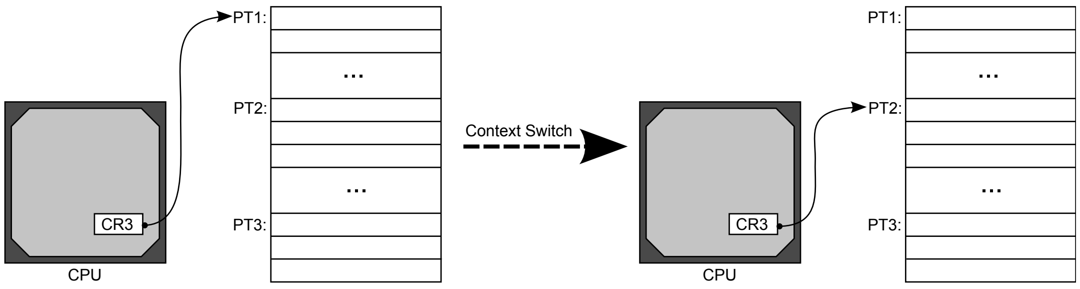


Figure 6.3: The CR3 contains the location of the top-level page directory for the process that is currently in context.

6.2.1 VM State Access

A very straightforward mechanism for VMI is simply a matter of checking and comparing CPU and/or memory state at regular intervals. The challenge in such an approach is determining which portions of the state are helpful. This section outlines exactly those portions of the state based on knowledge of the x86 architecture.

In addition to identifying those portions of state which are helpful, one must consider the trigger upon which this state is inspected. A timer may be the simplest solution. Although, event-based triggers may often be more appropriate. For example, if we wanted to use a context switch as the event trigger, we may use the mechanism suggested in Section 6.2.2.1.

6.2.1.1 Control Register 3 (CR3)

The CR3 register holds the address of the top-level page directory for the current context when paging is activated. That is, each process has a unique top-level page directory and the address of this page directory is stored in the CR3 register as is depicted in Figure 6.3. Since each process has a *unique* top-level page directory, the address of this directory acts a unique identifier. Hence, monitoring this register may be used to enumerate processes. The challenge here is determining when a process is in fact no longer running as opposed to simply idle. One may have to rely on heuristics and create a baseline for the running system to determine a reasonable idle time. The Lycosid system [37, 38] makes use of a exactly this mechanism to enumerate processes within a system.

6.2.1.2 IDTR and System Call MSRs

Monitoring the IDTR and the system call MSRs allow us to monitor for the placement of interrupt hooks at this level. Generally such a hook is placed within the system call mechanism, though hooks could be placed to catch any interrupt within the system. We will focus on monitoring for system call hooks, however the process described here for interrupt-based system call hooking may be used to monitor for any hooks within the interrupt system.

As mentioned in Section 6.1.1.1, there exist two primary system call mechanisms for system calls within the x86 architecture. The first method is to use the IDT to store the address of the system call handler and use the `int` instruction to perform the actual system call. The base address and size of the IDT are stored in the IDTR and the layout of the entries in the IDT (gate descriptors) are specified by the hardware architecture. These gate descriptors provide a logical address for each interrupt handler. This information allows us to monitor for any interrupt hooks (including system call hooks) at the IDTR and IDT level.

The second, more recent method involves MSRs which are responsible for handling fast switching to the system call handler. This fast system call mechanism is provided through the `SYSENTER` or `SYSCALL` instructions. Going forth we will refer to the Intel syntax (i. e., `SYSENTER`) as the technical differences of these two mechanisms are irrelevant here. The MSRs store, among other things, the offset for the system call handler.

In the case of system calls, we refer to the code that the IDT or the appropriate MSR point to as the *dispatcher* routine. We give it this distinction because this function does not "handle" the system call. Rather, it examines the calling conditions and refers to a data structure called the *system call table* in order to determine where the appropriate handling function is located, then hands the appropriate information over to this handler.

As mentioned in Sections 6.1.1 and 6.1.1.1, monitoring for hooks at the register or IDT level alone is not sufficient. We must monitor the integrity of all stages of the system call mechanism, i. e., the appropriate registers, the IDT, the system call dispatcher, the system call table, and the system call handlers. Since the location and layout of some of these functions and data structures are not dependent on the hardware architecture, some delivered knowledge will be necessary.

It is possible to combine such a derivative method with a delivery approach in which the size and layout of the dispatcher and handler functions and the system call table are delivered to the hypervisor. Here we have an example of rooting the semantic information in the hardware as described in Section 5.3.1, thus providing a solution with increased robustness against circumvention. Let us consider a simple example. We have the ability to monitor the IDTR and therefore the IDT. The address of our dispatcher lies within the IDT and the address of the system call table lies within that dispatching procedure. Finally, the addresses of the handlers are contained within the system call table. Even though we rely on delivered information (i. e., the location and layout of the dispatcher, handlers, and system call table), it is impossible for a malicious entity to manipulate anything along this chain without the change being evident from the hypervisor since this chain is rooted in the hardware (IDTR or MSRs). If the malicious entity alters the system call table, the change is detectable. If it manipulates the dispatcher to use a cloned system call table, the change is also detectable. This chain continues until we reach the hardware (in this case the IDTR) and any changes here are clearly detectable as well. At this point an attacker cannot resort to a level lower than the hardware.

Both Windows and Linux use system call mechanisms similar to the one described here. However, it is possible to execute the kernel in a separate virtual address space rather than in a dedicated region of other process's address space. In this case the mechanism would look slightly different, though a similar technique could be incorporated.

6.2.1.3 GDTR and LDTR

The Global Descriptor Table (GDT) and Local Descriptor Table (LDT) are data structures that are crucial to the segmentation mechanism in the x86 architecture and the GDTR and the LDTR are the registers that contain the location and size of GDT and LDT, respectively. The layout of these tables is specified by the hardware specifications and they may be used to regulate the segments that are accessible at various protection levels and facilitate the switch between these protection levels. Since these data structures play a part in the protection mechanisms of the x86 architecture, they may become a target for a malicious entity. Thus, monitoring them and their respective data structures for integrity may become crucial in a security solution.

In practice, monitoring these registers and their respective data structures has limited usage as OSs may (and often do) choose to effectively ignore this mechanism. That is, the OS may choose to provide isolation and protection by leveraging the paging mechanism and use the segmentation mechanism in a minimal way. For example, the Linux 2.6 kernel sets up all kernel and user segments such that the logical addresses are equivalent with linear addresses. This means that both protection levels share the same segments and this mechanism provides no protection at all. This leaves little for a malicious entity to abuse in this respect. Though, in systems which rely on segmentation for protection monitoring the integrity of these registers becomes crucial.

6.2.1.4 Virtual I/O

The VM must rely on the hypervisor for all input and output since the guest OS is provided virtual I/O devices by the hypervisor so that the physical device may be shared by multiple VMs or by the hypervisor itself. This can, of course, be leveraged by VMI mechanisms to monitor network traffic or keystrokes.

Performing such I/O monitoring is a matter of tapping into the virtual I/O device within the hypervisor and interpreting the data. When keystroke logging is the goal, for example, this would be a matter of tapping into the appropriate component of the hypervisor and logging the data stream to a file. Network traffic monitoring is another interesting application of this mechanism.

A simple example of taking advantage of network traffic monitoring is to perform network-based intrusion detection or firewall duties from the hypervisor. However, more interesting is to leverage the fact that the mechanism is performing its duties from the hypervisor. For example, Srivastava and Giffin [77] leverage this fact to present an application-aware firewall which is isolated from the running guest. Such an approach requires some delivered information, but is an excellent example of tapping into a virtual I/O device and combining this approach to come to a result that brings together the primary advantages of both host-based (i. e., application-awareness) and network-based (i. e., isolation) firewalls.

It is possible to take this one step further by combining the ability to tap into the network stream with a simple virtual CPU state inspection of the CR3 register. This register may be used to identify unique processes running in the VM as discussed in Section 6.2.1.1. This allows us to create network activity profiles for unique processes and would allow us to perform process-specific intrusion detection and firewall duties. That is, if a process is tagged as acting suspicious it may be blocked on a process granularity. This approach maintains isolation and relies completely on derived information and is therefore robust against circumvention techniques that might try to hinder application-aware solutions from correlating a network stream to an application correctly.

6.2.2 Virtualization Extensions

This section goes into depth considering how to best leverage several aspects of the virtualization extensions provided by both Intel (VT-x) and AMD (SVM) for VMI. Unless otherwise specified, these techniques are possible on Intel as well as AMD hardware which provide similar x86 virtualization extensions.

6.2.2.1 Context Switch Trapping

The virtualization extensions allow one to directly trap to the hypervisor on a context switch. This results in a fairly straightforward method for trapping context switches, however this method relies on the guest OS's usage of the TSS. It is possible for an OS to use the TSS mechanism in a limited way and perform software context switches. For example, the Linux 2.6 kernel makes limited use of the TSS, having a single TSS for each CPU. This results in no hardware context switches.

Since the above technique is not useful for all guest OSs, we discuss another technique that does not have the above restrictions and is also supported by both sets of CPU extensions. This technique traps writes to the CR3 register. The CR3 register is the control register which holds the offset of the current page directory as described in Section 6.2.1.1. Since each process must have its own page directory in OSs that make use of hardware-supported virtual memory, trapping writes to the CR3 register will result in a trap to the hypervisor on each context switch.

In fact, many hypervisors that use shadow page tables must trap all context switches (i. e., CR3 accesses) in order for the VM to run properly. In this case the work of trapping is already done, one must simply incorporate their intended action.

6.2.2.2 Virtual Machine Entry Trapping

As mentioned in Section 6.1.2.4, proof of concept malware does exist that demonstrates the ability to move the running OS into a VM, thus allowing the malware to perform introspection and take advantage of interposition [68]. In order to do this, a piece of malware must take advantage of the hardware extensions for virtualization. One must simply disallow any attempt to enter into a VM and react to this (e. g., logging, setting an alert, etc). Both Intel and AMD provide the ability to trap the entry instruction, so this can be done with relative ease.

6.2.2.3 System Interrupt Trapping

System interrupts are those interrupts which reside within the IDT at offsets from 0 to 31. Common system interrupts are page-fault exceptions and general protection faults. For a complete list we refer the reader to the Intel Developer Manuals [35]. It is possible to set a VM to trap to the hypervisor on any of these interrupts. That is, any action which causes a system interrupt may be trapped by the hypervisor.

This can be exploited by security applications in some cases by manipulating the guest from the hypervisor so that an event artificially causes a system interrupt which is then trapped by the hypervisor. Such approaches are discussed in Section 6.2.3.

6.2.2.4 User-defined Interrupt Trapping

User-defined interrupts use the same mechanism as system interrupts, though they must use an interrupt code within the range 32...255 (this also acts the offset into the IDT) and are caused by using the `int` instruction in addition to the advanced programmable interrupt controller (APIC). The APIC is responsible for timer interrupts and inter-processor interrupts, for example. System calls may be implemented in this way and thus may be trapped. The virtualization extensions provided by AMD allow native trapping of user-defined interrupts in the same way that system defined interrupts are trapped.

In order to trap these interrupts on Intel machines some extra steps must be taken. For a full discussion on this, see Section 6.2.3.3.

6.2.3 System Interrupt-enabled Mechanisms

As mentioned in Section 6.2.2.3, it is possible to leverage the fact that the x86 virtualization extensions allow us to cause the guest OS to trap to the hypervisor due to a system interrupt. Often times we may want to cause the VM to exit due to an event for which there exists no mechanism to cause a `VMEXIT` (i. e., trap to the hypervisor). In this case, it may be possible to manipulate the VM state in such a way that the target event will cause a system interrupt and set the hypervisor to trap this interrupt. The challenge here is how to manipulate the VM state so that the target event causes a trappable interrupt and how to minimize false positives. In this context, *false positives* are traps to the hypervisor that are due to our manipulation, but do not contribute to VMI and *natural* interrupts or exceptions are interrupts or exceptions which would have occurred regardless of our manipulation.

Many false positives will lead to significant performance overhead. It is therefore necessary to carefully craft such a mechanism in order to reduce the number of false positives. In addition, involving the hypervisor in natural interrupts when such intervention is not necessary will also lead to unwanted performance overhead. Both of these performance factors must be taken into account when considering such an approach.

The final challenge in such a mechanism is differentiating between natural interrupts and those caused by our manipulation of the VM state. This step will vary with each approach and is discussed in the appropriate subsections below.

6.2.3.1 Mitigating the Observer Effect

Changing the state of the VM leads to the possibility that an entity within the guest OS can detect or even deter our mechanism by looking for these changes and possibly altering this state. For this reason it is imperative to monitor the integrity of the corresponding state from the hypervisor. In the simplest case this can be achieved by monitoring the state at regular intervals. It may also be possible to use paging protection mechanisms in a hypervisor that makes use of shadow page tables since the page tables which are actually being used reside outside of the VM. Finally, in some cases it may be possible to deter a malicious entity from detecting the mechanism at all by trapping access to this state. For example, the x86 virtualization extensions allow us to trap accesses to most system registers and MSRs.

6.2.3.2 Page-Fault Exception

A page fault is an exception that is produced by the hardware when an exception occurs within the paging system of the architecture. We will concentrate on three specific causes of page faults which are helpful for VMI:

1. A page is requested for which the page-present flag is not set. We will refer to this as the Page Not Present (PNP) exception.
2. An attempt to write to a page for which the write flag is not set. We will refer to this as the Page Illegal Write (PIW) exception.
3. An attempt to execute operations on a page for which the non-executable flag is set. We will refer to this as the Page Illegal Execute (PIE) exception.

The PNP exception may be leveraged to indicate any type of access to a particular page belonging to a process. That is, the hypervisor may unset the page-present flag for a specific page whether the page is present or not and cause all page faults to trap to the hypervisor. The hypervisor must then intercept all page fault exceptions which were caused due to a PNP exception and for which the causing page is the page in question. The Ether system [17] uses such an approach to monitor system call activity. Rather than unsetting the page-present flag associated with the page in which the system call dispatcher is resident, it replaces the data in the MSR which holds the offset of the system call dispatcher with an offset into a page which is always set to be not present.

The PIW exception may be leveraged to indicate attempts to overwrite portions of memory. To make use of this the hypervisor must unset the write flags for the appropriate pages and handle these exceptions. This is a straightforward approach and works well when write protecting static data structures or code segments. In addition, the PIE exception may be leveraged to indicate attempts to execute certain portions of memory. These are straightforward methods as we are simply using the mechanism in the way they were intended although we are doing so from outside of the guest OS. Litty et al. [47, 46] make use of the PIE exception in yet another way. They use this mechanism to perform task enumeration. That is, they set the pages of a process to non-executable to determine whether the process is still running.

Changes made to force these interrupts are not visible to the guest OS in a system which uses shadow page tables as the shadow page tables are resident outside of the VM. This makes for a method which is robust against detection and manipulation attempts from within the guest VM.

This approach also seems as though it could be leveraged to perform code execution trapping. That is, trapping to the hypervisor when a portion of memory legitimately containing code is executed. This may be done to monitor the entrance into a function, for example. However, we caution the reader when using this approach for the following reasons: Such a method for code execution trapping will cause a trap on *each* instruction on the page causing many false positives; but more problematic, it forces the hypervisor to emulate all these instructions. This could solved by unsetting the non-executable flag after the first instruction is executed, though this leads us to two further problems. First, the hypervisor needs to continuously poll the VM state to determine whether it can reset the non-executable flag or we must set a time in which we are sure the function has completed execution. The second problem is that in this time another process could enter the same function without the hypervisor being made aware. Given these potential problems, the paging mechanism is not the proper approach for code execution trapping. For viable methods see Sections 6.2.3.5 and 6.2.3.4.

6.2.3.3 General Protection Exception

The general protection exception is produced for a wide variety of specific reasons all having to do with privilege protection. For the purposes of VMI we concentrate on the following five specific reasons:

1. An attempt to execute operation within a segment that is not executable. We will refer to this as the Segment Illegal Execution (SIE) Exception.

2. An attempt to write to a read-only data segment. We will refer to this as the Segment Illegal Write (SIW) Exception.
3. An attempt to read from a execute-only code segment. We will refer to this as the Segment Illegal Read (SIR) Exception.
4. Referencing an illegal, non-existent, or out-of-bounds entry within the IDT. We will refer to this as the Illegal IDT Reference (IIR) Exception.
5. Loading the CS register with a null segment selector. We will refer to this as the CS Null Segment Selector (CNSS) Exception.

Leveraging the first two exceptions, namely SIE and SIW, works almost identically to the mechanism described in Section 6.2.3.2 with regard to the PIE and PIW exceptions. The only difference is that here we are working within the segmentation mechanism of the hardware rather than the paging mechanism. This leads to a difference in granularity as the size of a segment is not static like the size of a page, though the VMI mechanism is almost identical except that here we set the hypervisor to trap and interpret a different exception.

The third cause for an exception is also due to the segmentation mechanism of the hardware. The VMI mechanism to leverage SIR exceptions is identical to that for the SIE and SIW exceptions, though here we have the ability to do something that the paging mechanism does not allow, namely trap attempted read access to code segments. This is useful for avoiding detection of VMI mechanisms that change code, as described in Section 6.2.3.4.

The issue with any VMI mechanism that leverages the hardware segmentation mechanism is that segmentation may not be properly used by modern OSs that use paging, as discussed in Section 6.2.1.3. Linux kernel 2.6, for example, uses this mechanism very minimally. So while these mechanisms may be useful in the correct environment, they rely on the guest OS's usage of segmentation.

The fourth cause for a general protection exception is the IIR exception. Unlike the above general protection exceptions, the IIR exception is independent of segmentation. This exception is useful for trapping user specified interrupts on Intel architectures. As explained in Section 6.2.2.4, the Intel virtualization extensions do not provide a mechanism for trapping interrupts greater than 31 (i. e., user specified interrupts). In order to induce a general protection exception, the hypervisor must alter the IDT entries for the interrupts that it wishes to intercept. This can be done in a number of ways. First, one can invalidate the IDT entry by unsetting the segment present flag in the call gate. Additionally, one can set the Descriptor Privilege Level (DPL) for each IDT gate descriptor to null. Finally, one can also reduce the size of the IDT by manipulating the IDTR, such that the entry falls outside of the IDT. In any case, any subsequent access to the IDT entry will result in a general protection exception which can be trapped by the hypervisor. It must be said that by setting the DPL to null, only in instances in which the Current Privilege Level (CPL) is greater than 0 (a user process) will the event be trapped.

Finally, the CNSS may be used in a very specific way to facilitate system call trapping when the `SYSENTER` instruction is being used. Among other things, the `SYSENTER` instruction will set the CS register based on the contents of SYSENTER_CS_MSR register. That is, saving the contents of the SYSENTER_CS_MSR in the hypervisor and setting it to null will result in an CNSS exception any time the `SYSENTER` instruction is executed. This exception can be trapped to the hypervisor and the instruction emulated, thus, allowing one to trap all system calls if the system uses the `SYSENTER` mechanism.

6.2.3.4 Invalid Opcode Exception

The invalid opcode exception provides a mechanism that allows the kernel to handle invalid opcodes should the CPU attempt to execute them. As with all the mechanisms described in Section 6.2.3, this mechanism relies on the hypervisor being set to catch and handle guest system exceptions.

This mechanism lends itself to performing code execution trapping. In order to do this, one must simply replace the opcode at the position one wishes to trap with an invalid opcode. Then, the hypervisor can distinguish this from natural invalid opcode exceptions by referring to the opcode that was intended to be executed. This does, however, require that the hypervisor emulate the replaced operation, which may be a challenge.

This is straightforward, though if a malicious entity where to know where to look, such a mechanism is easy to detect and even circumvent if further measures are not taken. For this reason, such an approach *must* be combined with a mechanism to write protect the appropriate area of memory in order to deter circumvention such as those described in Sections 6.2.3.2 and 6.2.3.3.

This exception has additional usefulness that is closely related to the CNSS exception discussed in Section 6.2.3.3. If the use of the `SYSCALL` instruction is deactivated in the Extended Feature Enable Register (EFER)—this is a matter of unsetting the SCE flag within the EFER—and the instruction is used, an invalid opcode exception is raised. As with the CNSS exception, this may be used to facilitate system call trapping, except that this method is useful if the system uses the `SYSCALL` instruction to perform system calls. The EFER must be manipulated from the hypervisor, making the `SYSCALL` instruction invalid. Then, the invalid opcode exception must be trapped to the hypervisor, the `SYSCALL` instruction emulated, and control returned to the guest.

6.2.3.5 Debug Exception

The x86 architecture provides debugging support through the use of debug registers and a Debug Exception. As part of this support the architecture allows one to define four linear addresses as addresses whose execution cause a debug exception. This exception can then be set to trap to the hypervisor. This gives us the opportunity to set four memory addresses whose execution will be trapped to the hypervisor. In addition, accesses to these registers can be trapped so that any attempt to change these values will be noticed.

This provides a good platform for performing code execution trapping, though it is restricted to exactly four addresses. In addition, it may interfere with debuggers which wish to make use of the hardware support within the guest. A simple solution is to simply ignore access to the debug registers from the guest OS while such a mechanism is active. We argue that VMI is generally performed on production systems where extensive debugging is not required and since software debuggers are not hindered, this is a feasible solution. In cases where these restrictions cause issues, Section 6.2.3.4 provides an alternative method for code execution trapping.

6.2.4 Countering Threats with Hardware Mechanisms

The mechanisms we described thus far are intended as a starting point for further derivative-based VMI applications. Table 6.1 summarizes which of these mechanisms can be used to counter which threat from Section 6.1.

6.3 Summary

Making use of derivative methods for VMI does not have to be a complex task and the advantages have been made clear. We presented a toolbox of building blocks that may be used to anchor view-generation in hardware. In addition to these building blocks, we provided a clear discussion as to how and when each of these should be used and for which threats they may be applicable.

We urge the reader to consider the advantages of derivative methods and to apply the building blocks presented here when realizing VMI applications. Building a derivative based VMI application is well worth the effort when possible.

Threat		*Security anchor*	
6.1.1	Interrupt hooks	6.2.1.2	IDTR and system call MSRs
		6.2.2.3	System interrupt trapping
		6.2.2.4	User-defined interrupt trapping
		6.2.3.2	Page fault exception
		6.2.3.3	General protection fault
6.1.2.1	Process hiding	6.2.2.1	Context switch trapping
		6.2.1.1	Control register 3 (CR3)
6.1.2.2	Module hiding	6.2.2.4	User-defined interrupt trapping
		6.2.3.2	Page fault exception
		6.2.3.3	General protection fault
		6.2.3.5	Debug exception
6.1.2.3	Network activity hiding	6.2.1.4	Virtual I/O
6.1.2.4	Virtual machine rootkits	6.2.2.2	Virtual machine entry trapping
6.1.3	Network-based attacks	6.2.1.4	Virtual I/O
6.1.4	Malicious processes	6.2.1.4	Virtual I/O
		6.2.2.4	User-defined interrupt trapping
		6.2.3.2	Page fault exception
		6.2.3.3	General protection fault
		6.2.3.4	Invalid opcode exception
		6.2.3.5	Debug exception

Table 6.1: Linking threats to suitable hardware mechanisms

Chapter 7

Derivative System Call Trapping

Having explored the various features of the x86 architecture in Chapter 6, we present a proof-of-concept system for system call trapping using a derivative approach. In this chapter, we describe the implementation of our prototype VMI framework, *Nitro*, for system call tracing and monitoring. As Nitro works from within the hypervisor, it is isolated from malicious activities within the VM and remains hidden from the guest OS. To our knowledge, Nitro is the first VMI-based system that supports all three system call mechanisms provided by the Intel x86 architecture and has been proven to work for Windows, Linux, 32-bit, and 64-bit guests. Moreover, due to its derivative nature, this framework is flexible enough to feasibly support any OS built upon the x86 architecture. Capturing and disseminating data is done in real-time without hindering usability of the guest, as our performance tests in Chapter 8 will show. Finally, Nitro is resistant to attempts at evasion due to hardware rooting. We will discuss the foundations of this approach, its implementation, and its properties throughout this chapter.

7.1 An Argument for System Calls

Most modern OSs have hardware-assisted protection mechanisms in place in order to isolate processes from each other and from the kernel. In addition, the OS must offer a controlled method for communication beyond this isolation. This method for communication most often takes the form of system calls. The mechanism for system calls is hardware-assisted and it allows an isolated process to request services from the kernel. On its own, a process cannot directly interact with the rest of the system. System calls are necessary to perform actions such as file operation, network communication, inter-process communication, etc. As system calls facilitate communication between the kernel and user space within an OS, they become very interesting from a security perspective.

Hofmeyr et al. [34] argue that system calls have the nice property that they are observable from the outside. That is, we do not have to necessarily understand what the process does internally to be able to collect the system calls. Each process may act as a black box and the system calls are emitted, observable data and by observing this we can reason about the normal behavior of a process.

In the following sections, we present a number of applications within the field of security for which system calls are used very successfully. We present them to motivate the fact that the information that system calls provide can be valuable in a variety of security applications.

Before we discuss these applications, it is important to note that some OSs wrap their system calls in system APIs. These system APIs allow for the underlying mechanism to change while the interface that is exposed to the application programmer remains constant and backward compatible. Some of the methods and applications discussed in this section make use of these APIs instead of system calls directly. As their function is the same we treat them as interchangeable.

7.1.1 Malware Detection

One interesting argument for using system calls for malware detection comes from the field of computer immune systems, which base their approaches on phenomena observed in natural immune systems. A prevalent theory in this field states that the natural immune system recognizes threats by distinguishing between protein fragments that belong to a normally functioning system and those that belong to invading cells [21]. These protein fragments that belong to a normally functioning system are referred to as “self”. The challenge becomes finding such a measure of “self” for processes. Several works suggest using system calls to this end [22, 34, 21]. Specifically, Forrest et al. [22] define the normal behavior of processes through short observed sequences of system calls that are then stored in a database. To detect abnormal behavior, this database is checked for deviations from this normal behavior during subsequent execution of the process and the result is a value that represents the percentage of mismatches with respect to all sequences observed. As this method is quite rigid (i. e., the system simply looks for legal sequences in the database), it was improved to include a measure of similarity between an observed sequence and a sequence present in the database [34]. This measure of similarity is in the form of a hamming distance between the two system call sequences. That is, instead of simply recording a mismatch, the improved system is able to take similarity to entries in the database into account. This is accomplished by finding the minimum hamming distance between the currently observed sequence and all sequences in the database. This minimum value is then taken for all observed sequences and the resulting maximum is normalized over the size of the sequence.

Another approach that aims to model the normal behavior for a single process is the work done by Kosoresow and Hofmeyr [42]. This approach works by building a deterministic finite automaton (DFA) to model the transition of states through sequences of system calls. That is, the states are divided into a start node, intermediate nodes, and one or more termination node(s), while the transitions are represented by edges that correspond to a sequence of system calls. With such a model, one can execute a process and follow the respective state transitions in the DFA. If one comes across a situation for which there is no legal state transition in the model, a deviation from the normal behavior has occurred. While very interesting, this approach suffers from the fact that one must build a DFA manually for each process one might want to observe. This is a difficult procedure especially for complex processes.

All the above approaches that are based in computer immune systems suffer from the fact that this model for “self” must be built for each individual process. That is, these approaches assume that a mechanism for detecting abnormalities must be tailored for each individual process. In the field of natural immune systems, this may be the case as our immune systems are tailored for our bodies, however individually tailoring such a system for each process would lead to a tremendous amount of overhead that would have to be readdressed for each process or for each new iteration of a process.

To move away from this, Xiao and Stibor make interesting use of probabilistic topic models [92]. They probabilistically assign system calls to topics based on co-occurrence. These topics are probabilistic groupings, however the authors show that their model nicely groups system call into intuitive groups (e. g., file I/O, memory management, process management, etc). In parallel, topic transition matrices are generated to model each trace. These topic transition matrices are then the basis for classification.

The most common machine learning approaches make use of the frequency of system calls in a trace. These approaches will either take into account the frequency of system calls directly [64] or make use of n-grams [43, 85]. A n-gram is simply a sequence of system calls of size n. Such approaches will work by splitting the system call trace into such n-grams, sometimes overlapping them, then counting the frequency of each n-gram. The resulting frequency distribution for each trace is then analyzed to learn those distributions that are indicative of malicious behavior and those that are indicative of benign behavior. This is generally achieved with the use of support vector machines (SVMs). For an in-depth look at SVMs, please refer to Section 9.2.

Finally, a more straightforward machine learning approach makes use of k-nearest neighbor classifiers [44]. A k-nearest neighbor classifier generally works by making use of a vector space model. This model models known documents (i. e., system call traces) with labels based on the occurrence and weight of words (i. e., system calls) in the document. This is often represented in a matrix where the rows represent documents, the columns represent words, and an entry represents the weight of the respective word in the respective document. For each new, unknown trace, the system calls within it are weighted and a similarity function is used to find the k most similar traces. The classifier then looks at the labels of these k nearest neighbors to determine under which label to classify the trace.

There are also further works that compare several machine learning methods for malware detection using system call traces [87]. Simply by looking at the wide range of methods that have been used to classify malware based on system calls one can see that system calls are very interesting from a malware detection perspective and will continue to be so.

7.1.2 Sandbox Environments

Related to malware detection, sandboxing environments remove the automation and allow an intelligent analyst to manually analyze malware samples. A critical piece of information in such an analysis is a system call trace of the process in question. This information among other data points provide the analyst with the information he or she need for a dynamic analysis. There exist several prevalent sandboxing environments that incorporate system call or API monitoring to create their reports [90, 10, 2, 25, 4]. These environments generally trap system calls in the same manner, by hooking the system call mechanism within the guest.

Another form of sandbox environments are high-interaction honeypots. Here the application is not a focused analysis of a single piece of malware as is often the goal in traditional sandbox environments, but rather a broad audit of an entire system. A honeypot is “an information system resource whose value lies in unauthorized or illicit use of that resource” [81]. That is, a honeypot is a system that is there to attract malicious entities for the purpose of their study. Trapping system calls aids in the study of the actions of these malicious entities [80].

7.1.3 Intrusion Prevention

While sandbox environments tend to passively monitor the activity of a process or system, in part, through system calls, intrusion prevention systems (IPSs) actively restrict certain actions based on some policy. Network-based IPSs are generally a more popular approach for intrusion prevention, however there are examples of in-guest IPSs that restrict the use of system calls based on some policy [27, 23, 60]. The method for trapping system calls is similar to the methods used in sandbox environments, the difference comes in the form of the policy engines used in these IPSs. That is, such approaches collect system calls in, generally, the same manner as sandbox environments, however IPSs make use of the system calls directly by potentially blocking them.

7.2 System Call Mechanisms in the Intel x86 Architecture

The system call interface is fundamental for any OS and is supported by the Intel x86 architecture through one of three mechanisms. Which of these mechanisms is used is left to the OS developers. Although the system call mechanisms were briefly described in Section 6.1.1, we will revisit each mechanism with a bit more detail. Unless otherwise stated, the information in the following sections is taken from the Intel Software Developer Manuals [35].

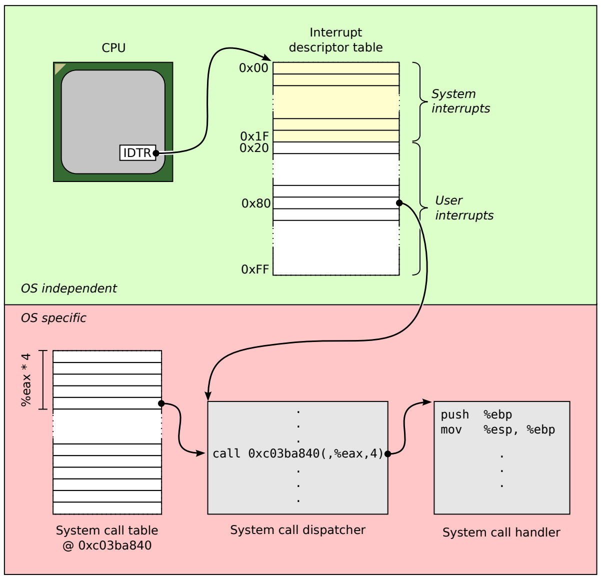


Figure 7.1: Relationship between the IDTR, the IDT, and the system call dispatcher

7.2.1 System Calls as an Interrupt

The x86 architecture handles interrupts through the IDT which contains gate descriptors for each interrupt. The IDT itself is stored in system memory, but the location and size of the IDT are stored in the IDTR. The gate descriptors within the IDT facilitate the jump to the appropriate handler functions. That is, when an interrupt occurs, the hardware consults the IDT via the IDTR to determine the location for the appropriate handler and continues execution there.

The IDT may have as many as 256 entries, each of which is 8 bytes long. The exact size of the IDT in bytes is stored in the IDTR along with the address at which the IDT resides. Of these entries, the first 32 entries are reserved for system interrupts. These system interrupts include division by zero exceptions, page faults, etc. Entries in the IDT beyond the 32nd entry may be defined by the kernel and are invoked either by the `int` instruction or through the Advanced Programmable Interrupt Controller (APIC). Of these two sources, the `int` instruction is interesting from a system call perspective and it accepts one argument. This argument determines an offset within the IDT. If an interrupt is requested at an offset beyond the bounds set by the IDTR, the result is a general protection fault.

This mechanism lends itself well to implementing system calls. To do this, a specific user-defined interrupt, e. g. 0x80, is defined as the system call interface by storing the location of the system call dispatcher in the IDT at position 0x80. Subsequently, a system call is invoked through the use of the `int` instruction with parameter 0x80. This relationship is depicted in Figure 7.1.

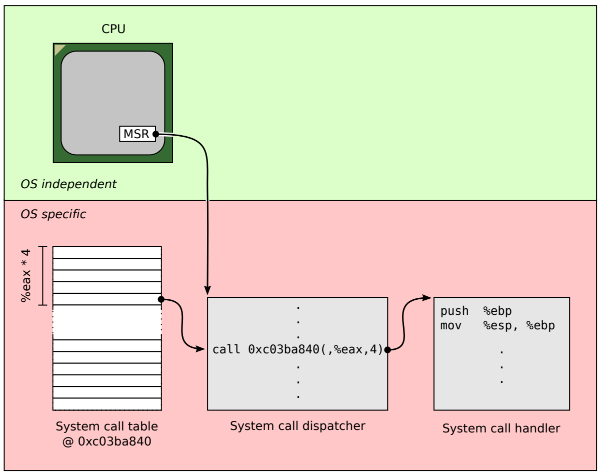


Figure 7.2: Relationship between the system call related MSR and the system call dispatcher

7.2.2 Fast System Calls

As the sophistication of operating systems grew, it became clear that the interrupt-based system call interface was often a performance bottleneck. This bottleneck results from having to consult the IDT before being able to pass control to the system call dispatcher. As system calls are a vital part of modern OSs and are frequently used, it became clear that a faster system call mechanism was needed. To address this, hardware manufacturers developed dedicated system call mechanisms. These mechanisms work by making use of MSRs to jump directly to the system call dispatcher when a specific instruction is called. Analogous, a corresponding return instruction restores control to the user process. More precisely, these MSRs contain the location of the system call dispatcher and a call to the proper instruction sets the Instruction Pointer (IP) based on these MSR values. This relationship is depicted in Figure 7.2.

One such mechanism makes use of the `SYSCALL` instruction along with its corresponding return instruction, `SYSRET`. As previously stated, this mechanism makes use of several MSRs (namely, STAR_MSR, CSTAR_MSR, and LSTAR_MSR). When the `SYSCALL` instruction is called, the IP is set based on the contents of the appropriate MSRs to handle the system call. Additionally, this mechanism can effectively be turned on and off by setting and unsetting the SCE flag in the Extended Feature Enable Register (EFER).

The second fast system call mechanism provided by the x86 architecture makes use of the `SYSENTER` and `SYSEXIT` instructions. This mechanism works in a similar fashion to the aforementioned mechanism, though different MSRs are used (i. e., SYSENTER_CS_MSR, SYSENTER_ESP_MSR, and SYSENTER_EIP_MSR). In addition to the IP being set, the value held within SYSENTER_CS_MSR is copied into the Code Segment (CS) register (this will become important in Section 7.3, when discussing the implementation of Nitro).

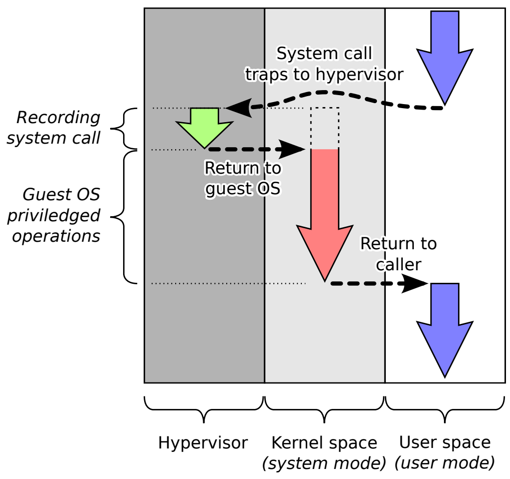

System call traps to hypervisor
Recording system call
Return to guest OS
Guest OS priviledged operations
Return to caller
Hypervisor
Kernel space (system mode)
User space (user mode)

Figure 7.3: Control flow of a system call that traps to the hypervisor

7.3 Implementation

This section describes the steps we took in implementing our prototype. Nitro is based upon the Linux Kernel Virtual Machine (KVM) [1]. KVM is split into two portions, namely a user application that is built upon QEMU [3] and a set of Linux kernel modules.

The user application portion of KVM provides the *QEMU monitor* which is a shell-like interface to the hypervisor. It provides general control over the VM. For example, it is possible to pause and resume the VM as well as to read out CPU registers using the monitor. We modified KVM by adding new commands to the monitor to control Nitro's features. That is, all Nitro commands are input via this monitor.

These commands are then sent to the kernel module portion of KVM through an I/O control interface. The majority of Nitro is implemented in these kernel modules. Finally, the output is realized by making use of netlink sockets [55]. That is, Nitro broadcasts the system call traces via a netlink socket and any process may listen for this broadcast and make use of Nitro's output.

7.3.1 VMI Mechanisms for Trapping System Calls

As mentioned in Section 6.2.3, it is often the case, especially for security mechanisms, that the hardware extensions do not support trapping the desired event. In these instances, we must indirectly induce a trap to the hypervisor. Finding these indirect methods for trapping desired events is often a challenge.

As it turns out, trapping to the hypervisor on the event of a system call is not supported in the Intel x86 architecture. In this case, we must find a way to indirectly cause the trap as discussed in Section 6.2.3. We do this by forcing system interrupts (e. g., page faults, general protection faults, etc) for which trapping is supported by the Intel Virtualization Extensions (VT-x). Hence, we have effectively created a mechanism for trapping system calls even though the hardware extensions do not natively support this. The resulting control flow is depicted in Figure 7.3. Since the three system call mechanisms are quite different in their nature, a unique trapping mechanism must be designed for each. These trapping mechanisms and their implementations are described below.

7.3.1.1 Interrupt-based System Calls.

Intel's VT-x extensions allow one to trap system interrupts (interrupts 0 to 31) to the hypervisor, but they do not provide a mechanism for trapping user-defined interrupts (interrupt 32 and above) which may be used for system calls.[1] This means that we must design a way to cause this user-defined interrupt to generate a system interrupt.

We can achieve this by creating a shadow IDT, that is, we copy out the guest's IDT into the hypervisor. We must then manipulate the IDTR and trap all write accesses to it, thus disallowing any further manipulation. As the IDT size value stored in the IDTR is added to the base address to get the offset of the last valid byte of the IDT, we can set this size to $32 \cdot 8 - 1 = 255$. This leaves all system interrupts unaffected, however all attempts at invoking a user-defined interrupt (i. e., interrupts greater than 31) will result in a general protection fault as the bounds of the IDT will have been exceeded. The advantage of this approach is that the IDT remains unaffected in memory, but is effectively ignored for user-defined interrupts.

The next step is to trap all general protection faults to the hypervisor, which the virtualization extensions support natively. However, we must still determine the difference between general protection faults that we generated and those that occur naturally[2]. This can be done by inspecting the current instruction and determining whether or not it is the `int` instruction and whether the interrupt number is greater than 31.

If we identify the exception as being natural, we inject this exception into the guest and allow it to continue. However, if we recognize the exception to be caused by a user-defined interrupt, we look at the interrupt number to determine whether we have trapped a system call. If this is the case, we collect data according to the rules specified for Nitro's data collection engine (see Section 7.3.3). In either case, the user-defined interrupt must be emulated using the shadow IDT that we copied out of the guest and hand control back to the guest OS.

7.3.1.2 SYSCALL-based System Calls.

System calls may also be implemented using the `SYSCALL` instruction and its analogue counterpart `SYSRET`. Both of these rely on a set of MSRs, namely STAR_MSR, CSTAR_MSR, and LSTAR_MSR. Exactly which of these registers is used depends on

[1]In contrast, AMD's SVM virtualization extensions do provide a mechanism for trapping user-defined interrupts.

[2]We refer to exceptions that are not caused by our changes as *natural* exceptions.

whether the guest OS is running in legacy, long, or compatibility mode. Additionally, this mechanism can effectively be turned on and off by setting and unsetting the SCE flag in the Extended Feature Enable Register (EFER). Making use of either `SYSCALL` or `SYSRET` with the SCE flag not set results in an invalid opcode exception.

Forcing this mechanism to cause a system interrupt is then a matter of unsetting the SCE flag and setting the hypervisor to trap all invalid opcode exceptions, which is natively supported by the virtualization extensions. Once control has passed to the hypervisor, we must once again differentiate between natural exceptions and those caused by our introspection. This is achieved by looking at the violating instruction and if this instruction is *not* either `SYSCALL` or `SYSRET`, we inject an invalid opcode exception into the guest OS and return control to it. However, if the violating instruction is, in fact, `SYSCALL`, Nitro collects the desired information, emulates this instruction, and returns control back to the guest OS.

In addition to emulating the `SYSCALL` instruction, Nitro must be capable of handling exceptions caused by the `SYSRET` instruction and emulating this instruction as well. This is due to the fact that the changes made to the EFER affect the `SYSRET` instruction in the same manner that they affect the `SYSCALL` instruction. Thus, use of the `SYSRET` instruction will also cause an invalid opcode exception and must be handled accordingly.

7.3.1.3 SYSENTER-based System Calls.

Similar to `SYSCALL` and `SYSRET`, the `SYSENTER` and `SYSEXIT` pair of instructions also rely on a set of MSRs, namely SYSENTER_CS_MSR, SYSENTER_ESP_MSR, and SYSENTER_EIP_MSR. The values in each of these MSRs are copied into specific system registers upon a call to `SYSENTER`. Specifically and most interesting for the development of Nitro, the value of the SYSENTER_CS_MSR is copied into the CS register when `SYSENTER` is executed and an attempt to load the CS register with a null value results in a general protection exception. Hence, causing a system interrupt is a matter of saving the current value of the SYSENTER_CS_MSR register in the hypervisor and loading it with a null value. This will cause each `SYSENTER` operation to attempt to load a null value into the CS register, thus causing a system interrupt that the hypervisor can trap.

Once the hypervisor has trapped a general protection exception, differentiating between natural and forced exceptions is once more a matter of checking the current instruction at the time of the exception. If we come across a natural exception, as with the previous system call mechanisms, we inject the exception into the guest OS and allow it to continue. In the case that we come across general protection exception and the current instruction is `SYSENTER`, we collect the relevant data, emulate the instruction using the saved value of the SYSENTER_CS_MSR, and return control to the guest OS.

As with the `SYSCALL`/`SYSRET`-based system call mechanism, the change that we make to the guest in order to induce a system interrupt also affects the `SYSEXIT` instruction. Consequently, we must also emulate this instruction.

7.3.2 Process Identification

It is always important to be able to determine which process produced a system call. This requires that we collect information which is unique to a process each time a system call is trapped. Nitro collects the value of the CR3 register along with the value of the first valid entry in the corresponding top-level page directory. This allows us to identify a process due to the fact that the value in the CR3 register (i. e., the address of the top-level page directory) is unique for a single, current process as described in Section 6.2.1.1.

However, we found that the CR3 value alone is, practically, not enough to uniquely identify processes over time. What we experienced through testing is that a newly created process may possess a top-level page directory that resides at the same location than the top-level page directory for a previously existing process. In our testing, this produced traces for which two subsequently occurring, individual processes were marked with the same identifier as the CR3 register contained the same value for both processes. In order to address this, we also consider the first valid entry in the corresponding top-level page directory in order to create a truly unique identifier.

7.3.3 Collection of System Call Data

In our experience, different applications for system call traces depend on varying amounts of information. In some cases a simple sequence of system call numbers without arguments may suffice, while other scenarios may require detailed information including register values, stack-based arguments, and return values from a small subset of system calls. As we cannot foresee every guest OS type and possible application of system call tracing, Nitro does not deliver a fixed set of data per system call. Instead, it allows the user to define flexible rules to control the data collection during system call trapping in a fine-grained manner. Nitro implements two modes of operation, namely system call tracing and rules-based system call monitoring.

7.3.3.1 System Call Tracing

System call tracing represents the broad and shallow approach to information gathering. A system call trace focuses on which system calls were called by what process and in which order with limited focus on the data that was transfered. When performing system call tracing, Nitro allows the user to define where the system call number is stored (generally in the EAX register) and collects this information along with a process identifier as described in Section 7.3.2. This information is often enough for certain machine learning techniques used for detection of malware or malicious behavior in processes [22, 42, 34]. The work of Rieck et al. also indicates that considering the arguments for system calls in addition to the system call numbers does not significantly improve the classification accuracy of malware classes [65].

rule	::=	**add_scmon_rule** <condition> <location> <action>
condition	::=	<register> <value>
location	::=	<register> <offset>
register	::=	rax | rbx | rcx | rdx | rsp | rbp | rsi | rdi
value	::=	[0,4294967295]
offset	::=	[-2147483648,2147483647]
action	::=	hex | int | uint | derefhex | derefint | derefuint | derefstr

Figure 7.4: Nitro rules in Backus-Naur Form.

7.3.3.2 Rules-based System Call Monitoring

On the other hand, system call monitoring represents the narrow and deep approach to information gathering. When monitoring system calls, system call arguments or even dereferenced memory variables pointed to by arguments may be of interest. Nitro allows a user to define what information he or she is interested in upon any system call(s) he or she defines. This is done is a rules-based manner. These rules are flexible enough to allow one to monitor the contents of a register or even to dereference registers and look into memory. This allows one to extract the arguments passed to the system call, whether they are passed in registers or within the stack. The syntax of such a rule takes the following format:

`add_scmon_rule CONDITION_REG CONDITION_VAL ACTION_REG OFFSET ACTION`,

where `CONDITION_REG` contains the name of the register that should be tested to determine whether further information should be output, `CONDITION_VAL` contains the value the `CONDITION_REG` should contain in order for further information to be output, `ACTION_REG` contains the name of the register that contains the base value we are interested in, `OFFSET` contains the offset (positive or negative) from the `ACTION_REG` for the data we are interested in, and `ACTION` defines the format the output should take, this includes actions that result in printing or dereferencing the data as hexadecimal, integer, unsigned integer, or string. We provide a description of the rules in Backus-Naur Form in Figure 7.4. As an example, it is easy to specify a rule that dereferences and outputs the string being written every time a user process makes use of the write system call within a Linux guest. This rule would look as follows:

```
add_scmon_rule rax 4 rcx 0 derefstr
```

This rules-based method of requesting information makes Nitro very flexible and contributes to its OS agnostic nature.

```
Jun 20 17:58:20: sys_write: unsigned int fd, const char
     __user *buf, size_t count
  fd: unsigned int: 0x3 $\rightarrow$ (socket) $\rightarrow$ [...]: (SOCK_STREAM)
     flags: ()
  buf: const char __user*: 0x7FFF702FF320 $\rightarrow$
    buffer content hex (of size 107):
      47 45 54 20 2f 20 48 54 54 50 2f 31 2e 30 da 55 73 65
      72 2d 41 67 65 6e 74 3a 20 57 67 65 74 2f 31 2e 31 32
      20 28 6c 69 6e 75 78 2d 67 6e 75 29 da 41 63 63 65 70
      74 3a 20 2a 2f 2a da 48 6f 73 74 3a 20 67 6f 6f 67 6c
      65 2e 64 65 da 43 6f 6e 6e 65 63 74 69 6f 6e 3a 20 4b
      65 65 70 2d 41 6c 69 76 65 da da
    buffer content string:
      GET / HTTP/1.0
      User-Agent: Wget/1.12 (linux-gnu)
      Accept: */*
      Host: google.de
      Connection: Keep-Alive
  count: size_t: 0x6B
```

Figure 7.5: This output represents a combination of Nitro and InSight [69, 70]. Nitro facilitates the trapping of the system call and delivers the register values (e. g., the file descriptor number), while Insight is used to determine what these values mean in context of the guest system (e. g., the file descriptor is attached to a stream socket).

Keeping the design goals for Nitro in mind, we collect only information whose location and format is defined by the hardware specifications or for which a rule is specified. However, the flexible design of Nitro allows an easy incorporation of guest OS specific knowledge in order to collect additional information about the calling process. We have successfully combined Nitro with other projects such as InSight [69, 70] to include information such as process and user IDs into the output. However, we keep these projects separate in order to keep Nitro as simple and flexible as possible. This allows Nitro to be applicable in a greater range of applications. When combined with InSight, we are able to produce output as shown in Figure 7.5. This provides additional guest OS specific information, such as the type of descriptor being written to, while allowing Nitro to remain applicable of a wide range of guest OSs.

7.4 Properties

In this section we take another look at the properties that Nitro possesses. In doing so we revisit the properties discussed in Chapters 5 and 3. In reviewing the outlined properties below, one might notice that the binding property and the 'inspection of suspended VM' properties are not explicitly covered.

The binding property is not explicitly discussed as it is encompassed by evasion-resistance. That is, the binding property is the basis for hardware rooting which is necessary for evasion-resistance. Hardware rooting takes advantage of the binding property in that the hardware anchor is subject to the binding property. The binding property states that the function of the hardware anchor is bound to assumptions that will not change at run-time. As we discuss the evasion-resistant nature of Nitro, we do not explicitly discuss the binding property.

The 'inspection of suspended VM' property discussed in Chapter 3 is also not discussed. This is due to the fact that Nitro performs event-based monitoring which, by its nature, cannot be performed on a suspended machine. Although derivative methods are fundamentally capable of inspecting suspended machines, it becomes difficult to monitor system calls on an inert machine. Therefore, this property is not further discussed in conjunction with Nitro.

7.4.1 Isolation

While isolation is almost entirely afforded through virtualization, there are still some steps Nitro must take to ensure complete isolation. This comes in the form of protecting those portions of the guest state that Nitro manipulates. If we were not able to protect those portions of the state against malicious entities who wished to revert our changes, we would lose isolation. That is, these malicious entities would be able to "turn off" Nitro from within the guest.

As presented in Section 7.3, Nitro manipulates the EFER, the SYSENTER_CS_MSR, and the IDTR. Therefore, these registers must be protected such that their values cannot simply be reverted. Luckily the virtualization extensions from Intel provide us with mechanisms to trap write accesses to these registers. Using this mechanism we can simply block writes in the simplest case, thereby maintaining isolation.

7.4.2 Guest OS Portability

Nitro is guest OS portable due to the fact that all three mechanisms described in Section 7.3 make sole use of hardware knowledge. This allows the mechanisms to work for any guest OS that is compatible with the Intel x86 architecture. The IDTR and IDT as well as all the involved MSRs and their uses are specified by the hardware architecture and must be used in the way specified. That is, any guest OS must use these hardware mechanisms according to the specifications regardless of the guest OS.

In order to maintain guest OS portability in light of the fact that how information is passed between kernel and user space is left to the OS designer, Nitro provides the flexibility of the rules-based system described in Section 7.3.3. That is, the user can control which data is collected by specifying rules at run-time. This allows Nitro to be useful across all guest OSs by simply changing the rule set. In our experiments, we collected system call traces from Windows 7 (64-bit), Windows XP (32-bit and 64-bit), and Ubuntu Linux (32 and 64-bit) simply by changing the rule set each time.

7.4.3 Evasion-resistance

We make the reasonable assumption that the hypervisor itself is secure. In addition, any components that reside within the hypervisor are safe from attacks originating from within a guest OS due to the hypervisor's isolation property.

While we have the aforementioned assumption with regard to the hypervisor itself, this alone is not enough. This is due to the fact that our VMI mechanisms make changes to the state of the guest VM. These state changes are clearly not protected by the isolation property as they take place within the guest OS itself. A malicious entity might simply revert the changes we made to the system state to circumvent our security mechanisms. For this reason we took special care to make sure that Nitro is evasion-resistant.

As stated in Section 5.3.3, evasion-resistance requires that the VMI mechanism is rooted in hardware and that each involved piece of VM state is protected against manipulation. Since the VMI mechanisms for the fast system call mechanisms and the interrupt-based mechanism differ slightly in this regard, they are discussed separately in the following.

7.4.3.1 Protecting Fast System Call Trapping Mechanisms

In order to achieve an evasion-resistant VMI mechanism for fast system calls, it is rooted in either the SYSENTER_CS_MSR (`SYSENTER`-based) or the EFER (`SYSCALL`-based) as discussed in Section 5.3.3. In addition, the VMI mechanism may protect each of these registers from manipulation, which is directly supported by the virtualization extensions provided. This is enough to achieve evasion-resistance because Nitro's manipulation of the system call mechanisms are constrained to these registers. That is, due to the changes we made to the system, all fast system calls are trapped to the hypervisor and there is no way for a malicious entity to circumvent this without making changes to exactly those parts of the system that Nitro protects. Hence, this approach is both rooted in hardware and protects all involved pieces of VM state, resulting in an evasion-resistant mechanism.

7.4.3.2 Protecting Interrupt-based System Call Trapping Mechanisms

Making the interrupt-based system call traps evasion-resistant is similar to the method described for fast system calls with one additional step. The mechanism is rooted in the IDTR and this register is protected against malicious manipulation by the hypervisor. In addition, a shadow copy of the original IDT is created within the hypervisor at boot time. This already is enough to achieve evasion-resistance as the changes to the guest OS are limited to this register. In addition, only the shadow IDT is referred to for each user-defined interrupt. That is, any changes to the IDT within the guest OS do not affect the ability to trap user-defined interrupts. In order to hinder this, a malicious entity would have to manipulate the IDTR directly or the shadow IDT within the hypervisor, both of which are protected with the help of the virtualization extensions.

7.4.4 Mitigating the Observer Effect

As stated in Section 5.4, mitigating the observer effect is an important aspect of VMI if the goal is observation. As Nitro is a tool that would be well suited for observation, we discuss steps that can be taken to mitigate the observer effect.

The most straightforward of such steps is to make sure that the hypervisor reports the original values for all registers Nitro altered. The altered registers include the MSRs involved with the fast system calls and the IDTR. Luckily, these are registers for which the virtualization extensions support trapping reads and writes. So, this step requires trapping reads to these registers and returning the masqueraded (i. e., original) values rather than the real (i. e., manipulated) values.

In addition to read accesses, we must consider write accesses to these registers. Simply ignoring writes could easily result in detection. However, one could save all written values in the hypervisor and return these values whenever a read access occurs, all the while using the unchanged real value for system calls. This might fool naive methods of detection, however a clever entity might try to observe changes in behavior based on these writes and detect the system through such means. For example, a crass method for detection might be to write an invalid value into such a register and observe how the system reacts. If the system does not crash and the value of the register continues to reflect the invalid input, one can assume that the hypervisor is involved.

Additionally, one could attempt to accept all writes to these registers so long as we are still able to trap the particular system call. This has different implications for each affected register. First, we consider the EFER. In this case, we set the single SCE flag (i. e., bit) within the EFER for our system call trapping, so a change to other flags does not concern our implementation. If an entity within the guest were to manipulate the SCE flag, we could continue to keep the real flag set to 0 and trap all uses of the `SYSCALL` and `SYSRET` instructions. In the hypervisor, we can then use the masqueraded value of the SCE flag to determine whether we should carry out a system call or inject an invalid opcode exception. Next, we consider the implications this has for the `SYSENTER_CS_MSR` register. This is quite straightforward in that we can keep the real value set to 0 and simply use the masqueraded value in the hypervisor each time a system call is trapped. Finally, writes to the IDTR can be handled in a similar fashion. We maintain the real value unchanged, but use the masqueraded value for checking the size of the IDT when system calls are handled in the hypervisor. However, we must still consider the possibility that an entity changes the location of the IDT or values within the IDT itself.

In our implementation, the shadow IDT is used for interrupt-based system calls. However, if an entity within the guest changes the location of the IDT, we cannot be sure that the IDT in the new location is identical to the original IDT (i. e., the shadow IDT). In this case, we would need to overwrite our shadow IDT with the guest's IDT in the new location. This covers the case that an entity makes use of the IDTR to detect the presence of Nitro, however detection is still possible in that an entity makes a change to the IDT itself and observes the effects. In this case, one could forgo the shadow IDT and refer to the IDT within the guest OS each time a user-defined interrupt is invoked. While this would incur additional performance overhead, it would lead to the possibility of a malicious entity making changes to the IDT that take effect, however all user-defined interrupts will still be trapped.

7.4.5 Performance

In most VMI-based mechanisms, performance overhead becomes a concern. For this reason, it is important to keep unnecessary traps to the hypervisor at an absolute minimum. In all the mechanisms described in Section 7.3, we make use of a system interrupt to facilitate the trap to the hypervisor due to the fact that system calls are not natively trappable.

For our implementation, we looked at the individual system call mechanisms and determined all system interrupts that each system call mechanism could be made to produce and how to induce them. We then inspected all feasible solutions and considered them in terms of their impact on performance. For example, all three system call mechanisms can be made to produce a page-fault, however we passed on this for two reasons. First, page faults occur often (relative to other system interrupts) in regular system activity. This means that each page fault would result in a costly trap to the hypervisor to distinguish between forced and natural page faults, most of which would be natural. Second, this would essentially nullify any performance improvement that comes from using Extended Page Tables or Nested Page Tables.[3] In general, we strove for a system interrupt that occurs *infrequently* during normal operation and one whose use would not counteract performance enhancements in other parts of the system.

For a detailed performance evaluation see Chapter 8.

7.5 Remaining Attack Vectors

We have argued in Section 7.4.3, that, given the registers in question are protected, Nitro is resistant to any evasion attempts. Specifically, we mean that there is nothing a malicious entity can do such that communication through one of the three system call mechanisms we describe in Section 7.2 will not be trapped to the hypervisor. There may, however, remain other mechanisms by which a user space entity can communicate with the kernel or vice versa. The simplest of such mechanisms may be facilitated by the fact that the kernel can, by its right, read from and write to anywhere in memory. That is, the user space entity could simply write to or read from a predetermined part of its address space and the kernel could reciprocate. Such communication cannot be detected or trapped with Nitro.

[3]These are hardware extensions implemented by Intel and AMD, respectively, to counter the performance degradation caused by using shadow page tables.

There have been further suggestions for obfuscating system calls or making use of other mechanisms for system calls, for example, by embedding one system call in another [79] or by using other system interrupts to facilitate system calls [49]. In the case of embedding one system call within another, Nitro will still trap this communication to the hypervisor and deliver all requested information. Nitro makes no effort to interpret the system calls, but rather to deliver the information and in such a situation the communication will be trapped and delivered due to the fact that, regardless of the obfuscation, system calls are still being used.

In the case of making use of other system interrupts to facilitate system calls, this communication will not be trapped by Nitro as it is designed to trap system call interrupts. Such communication could, however, be trapped using a method similar to that of Nitro. In fact, it would be more straightforward in that system interrupts are natively trappable both on Intel and AMD hardware. However, such communication could take place without Nitro being aware.

7.6 Summary

We have shown that Nitro is a powerful and flexible tool for system call tracing and monitoring. It supports all three system call mechanisms provided by Intel's x86 architecture for both 32-bit and 64-bit environments. In fact, we have successfully collected system call traces with Nitro for Windows, Linux, 32-bit, and 64-bit guests and we are confident that it will perform equally well for a variety of additional guest OSs. Further, all of the VMI mechanisms presented have been shown to be evasion-resistant. That is, these mechanisms cannot be manipulated in a way which allows a malicious entity to circumvent system call tracing or monitoring. Its flexible and secure nature allows Nitro to be used effectively in a variety of applications, such as machine learning approaches to malware detection, honeypot monitoring, as well as sandboxing environments. In the next chapter, we go on to strengthen the argument for Nitro by presenting a thorough set of performance tests which include a comparison to a similar system, Ether [17].

Chapter 8

System Call Trapping Evaluation

In this chapter, we present our general performance testing results for all guest OSs tested with Nitro, which include: Windows 7 (64-bit), Windows XP SP2 (32-bit), Ubuntu Linux 9.04 Server (32-bit), and Ubuntu Linux 9.04 Server (64-bit). We tested such a large variety of guest OSs to demonstrate the power and flexibility of Nitro. The tested OSs range from client to server OSs as well 32-bit to 64-bit OSs. Additionally, we show that Nitro works with commercially popular OSs as well.

These tests were performed by running benchmarks on the guest OSs once with Nitro disabled, then once with Nitro enabled and comparing these results to compute the performance degradation incurred on each platform. While the results themselves are of interest, we focus primarily on the amount of degradation observed because the degradation is a strong indicator for the overhead incurred by our system call tracing.

Throughout these tests, we were not only careful to test as many guest OSs as possible, but also to test each system call mechanism. That is, we present tests that measure the performance of our mechanism for `SYSENTER`-based, `SYSCALL`-based, and interrupt-based system calls. Not all OSs support each of these mechanisms, however we make sure to present at least one test for each mechanism.

The following subsections present our results. All scores and times presented are a mean over three scores or runs.

8.1 Windows XP

These tests were performed on an Intel Core 2 Duo processor at 2.4 GHz with 2 GB of RAM. In testing the Windows XP guest OS we made use of two commercial benchmarking products, namely PCMark05 from Futuremark and PerformanceTest from PassMark.[1] These tools perform various CPU, memory, disk drive, and graphics tests. Each

[1]Available from `http://www.futuremark.com/products/pcmark05/` and `http://www.passmark.com/products/pt.htm`, respectively.

Benchmark	*Tracing disabled*	*Tracing enabled*	*Degradation*
HTML Render [pg/s]	2.826	2.034	28.04%
File Decryption [MB/s]	64.697	64.654	0.07%
HDD [MB/s]	46.545	9.726	79.10%
Text Edit [pg/s]	84.743	40.032	52.76%
Image Decompression [MPix/s]	33.364	33.103	0.78%
File Compression [MB/s]	2.744	2.741	0.10%
File Encryption [MB/s]	15.853	15.826	0.17%
Virus Scan [MB/s]	314.118	155.718	50.43%
Mem. Latency [MemAcc/s]	6.231	3.782	39.30%
PerformanceTest Overall [score]	628.700	540.260	14.07%

Table 8.1: Windows XP SP2 (32-bit) performance results with *SYSENTER-based system calls* on KVM/Nitro.

make heavy use of system calls as is evidenced by the output of Nitro.

The standard deviation of all tests were negligible, except for the “HDD” and “Virus Scan” tests where we observed standard deviations of 6.3 and 61.0, respectively. We hypothesize that this is due to the fact that these are both disk I/O intensive tests. In any case, we present these results for the sake of completeness, however, due to their high deviation from the mean, we do not draw any conclusions from these values.

It is interesting to note that across the set of tests the degradation varies greatly from benchmark to benchmark, however the benchmarks with the lowest degradation ($< 10\%$) all perform some sort of compression, decompression, encryption, or decryption. Such functions are highly arithmetic and perform relatively few system calls since these arithmetic operations do not require OS support. We believe that this is the reason for the variation in degradation across the benchmarks. While the PCMark05 tests (the first nine benchmarks in Table 8.1) are great for identifying to which degree an operation is affected by overhead in the system call mechanism, we feel that the results delivered by PerformanceTest give a better overall impression of the performance degradation in the guest OS as a whole.

8.2 Windows 7

These tests were performed on a an Intel i7 processor at 3.4 GHz with 8 GB of RAM using the PerformanceTest benchmarking suite. We forgo the use of PCMark05 as it is not fully supported by the 64-bit environment that Windows 7 provides. As can be seen in Table 8.2, the PerformanceTest benchmarking suite gives a nice overview of the performance and the degradation incurred through the use of Nitro.

Benchmark	*Tracing disabled*	*Tracing enabled*	*Degradation*
CPU [score]	1747.6	1585.1	9.30%
Graphics [score]	708.9	407.1	42.57%
Memory [score]	1443	1238.7	14.16%
Disk [score]	608.5	346.2	43.10%
Overall [score]	1159.0	872.9	24.68%

Table 8.2: Windows 7 (64-bit) performance results with *SYSCALL-based system calls* on KVM/Nitro.

	Apache Compile Results		
Linux Guest OS	*Tracing disabled*	*Tracing enabled*	*Degradation*
32-bit Interrupt-based	75.2s	92.2s	22.65%
32-bit `SYSENTER`-based	72.8s	95.3s	30.89%
64-bit `SYSCALL`-based	65.9s	81.7s	23.92%

Table 8.3: Ubuntu Linux 9.04 Server performance results on KVM/Nitro.

As was the case for Windows XP, we see that the tests that are CPU/memory intensive ("CPU" and "Memory" benchmarks) incur lesser degradation than operations that need OS support ("Graphics" and "Disk" benchmarks). This is, again, due to the fact that the CPU/memory intensive operations are, by their nature, not system call intensive. We draw the readers attention to the "Overall" score degradation as this gives the best indication of overall system performance. While the degradation is greater than that of the Windows XP guest it is still very much in an acceptable range and is still very functional from a user perspective.

8.3 Ubuntu Linux

For testing all Linux guest OSs we created a script that makes use of the "time" command in Linux. Using this utility we measured the compile time of the Apache web server 2.2.22 on an Intel i7 processor at 3.4 GHz with 8 GB of RAM. The time utility makes use of the hardware clock and we verified beforehand that the hardware clock within the VM is consistent with the host system's hardware clock. We used this as a benchmark as it is resource intensive enough to show performance degradation and makes extensive use of system calls as is evidenced by the output of Nitro.

Presented in Table 8.3 are the test results when performed on a Ubuntu Linux 9.04

Server (32-bit) guest OS. We manipulated the virtual hardware in order to be able to report results for interrupt-based and SYSENTER-based system call mechanisms. That is, by default, Ubuntu 9.04 will attempt to make use of a fast system call mechanism. However, when these mechanisms are not present, the kernel is capable of falling back to the interrupt-based method for system calls. We simply manipulated the hardware such that the guest would act as though the fast system call mechanisms were not present.

The testing process we used for a 64-bit Linux guest OS is identical to the processes we used for the 32-bit Linux guest with the obvious exception that we use Ubuntu Linux 9.04 Server (64-bit). One noteworthy difference between the 32-bit and 64-bit version of this operating system is that the 64-bit version makes use of the SYSCALL-based system call mechanism. These results are also presented in Table 8.3. Comparing the degradation of this guest to its 32-bit counterpart reveals that the degradation for all three tests are comparable.

Considering the results further, we notice that the guest OS making use of the interrupt-based system call mechanism incurs less degradation than the guest OSs making use of the SYSENTER- and SYSCALL-based system call mechanisms. This is interesting as the interrupt-based mechanism is far more complex than the fast system call mechanisms and the emulation in the hypervisor is accordingly more complex and therefore takes more time. Based on this alone, one would expect that the guest making use of the interrupt-based mechanism would incur a greater degradation. However, one must keep in mind that the trapping mechanisms for the fast system calls presented in Section 7.3.1 require that the entrance instruction as well as the return instruction be trapped to the hypervisor. This leads to two traps per system call when the guest makes use of the fast system call mechanisms. So, while the interrupt-based mechanism is more complex and likely does incur a greater overhead when compared one-to-one, this is offset by the fact that the fast system call trapping requires two traps per system call as opposed to one.

8.4 Comparison

We choose to compare our system to the Ether system [17] because Ether is the only other system to our knowledge that supports some forms of system call tracing using VMI without having to install drivers or modules in the guest OS (some others exist, though they base their system call trapping on Ether's approach [53, 78, 91]). Both in function and performance, Nitro surpasses Ether with regard to system call tracing and monitoring. In this section we discuss these differences in further detail.

8.4.1 Functional Differences

The largest functional difference between Ether and Nitro is the hypervisor that they are built upon. Ether builds upon the Xen hypervisor, while Nitro builds upon KVM. Nitro and Ether's system call tracing mechanism are similar in respect to the output

Benchmark	*Xen/Ether (SYSENTER)*			*KVM/Nitro (SYSENTER)*		
	Tracing disabled	*Tracing enabled*	*Degradation*	*Tracing disabled*	*Tracing enabled*	*Degradation*
HTML Render [pg/s]	3.277	0.598	81.74%	2.826	**2.034**	**28.04%**
File Decryption [MB/s]	65.561	64.561	1.53%	64.697	**64.654**	**0.07%**
HDD [MB/s]	45.198	7.215	84.04%	46.545	**9.726**	**79.10%**
Text Edit [pg/s]	89.066	17.246	80.64%	84.743	**40.032**	**52.76%**
Image Decompression [MPix/s]	33.856	32.951	2.67%	33.364	**33.103**	**0.78%**
File Compression [MB/s]	2.737	2.677	2.19%	2.744	**2.741**	**0.10%**
File Encryption [MB/s]	15.821	15.515	1.94%	15.853	**15.826**	**0.17%**
Virus Scan [MB/s]	333.988	85.307	74.46%	314.118	**155.718**	**50.43%**
Mem. Latency [MemAcc/s]	6.735	3.580	46.84%	6.231	**3.782**	**39.30%**
PerformanceTest [score]	586.500	383.020	34.69%	628.700	**540.260**	**14.07%**

Table 8.4: Windows XP SP2 (32-bit) performance comparison between KVM/Nitro and Xen/Ether.

they provide, though Ether's output is guest OS specific. That is, the output is Windows specific. Despite this, we tried Ether on further guest OSs in order to determine whether the underlying mechanism may be used for other guest OSs.

We tested both systems for functionality on Windows XP SP2 (32-bit), Ubuntu Linux 9.04 Server (32-bit), and Ubuntu Linux 9.04 Server (64-bit). Nitro proved functional on all tested platforms, while Ether proved 100% functional only on Windows XP SP2. While we expected Ether to be functional on Ubuntu Linux 9.04 Server (32-bit) because this OS uses the same system call mechanism as Windows XP SP2, the guest OS became very unstable and was not usable from a user's perspective when system call tracing was enabled. Finally, Ether was unable to provide any system call information for Ubuntu Linux 9.04 Server (64-bit) although the guest OS continued to run without issue. We believe that this is due to the fact that Ether fails to consider the `SYSCALL/SYSRET` mechanism for system calls completely. We see this as a major detractor as this limits the number of guest OSs for which system call tracing or monitoring will work. In addition, this opens up opportunities to evade Ether.

8.4.2 Performance Comparison

We performed all presented tests on an Intel Core 2 Duo processor at 2.4 GHz with 2 GB of RAM. In addition, we used the same Windows XP SP2 (32-bit) image for all tests to ensure the consistency of the guest OS. Finally, we used the recommended Xen 3.1.0 as hypervisor for Ether. It is also important to note that we chose the specific benchmarks out of the PCMark05 suite due to the fact that these are the same benchmarks that Ether's authors used when testing their system originally. We felt it was important to

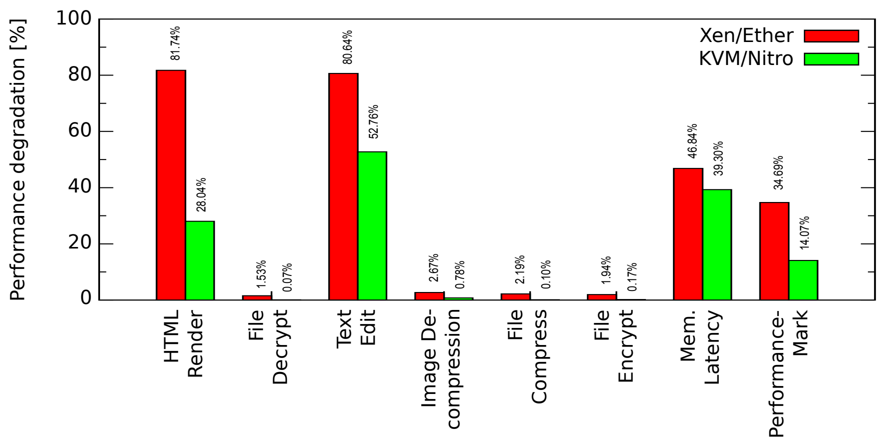


Figure 8.1: Performance degradation of Xen/Ether and KVM/Nitro when tracing a Windows XP SP2 guest OS (ref. Table 8.4).

include the same set of tests in the interest of equity.

Nitro and Ether are based on two different hypervisors, namely KVM and Xen. As our intentions were not to compare the performance between KVM and Xen but to compare the efficiency of the different implementations for system call tracing, we look at the relative performance degradation between the unmodified version of KVM and Nitro and compare this to the relative performance degradation between the unmodified version of Xen and Ether. This way we can measure the performance overhead incurred by each VMI implementation and present a fair comparison of Nitro and Ether.

As Ether only worked correctly for a Windows XP SP2 guest OS, we were only able to compare the performance of Ether and Nitro on this guest. These results are presented in Table 8.4 and Figure 8.1. Again, we do not draw any conclusions from the 'HDD' and 'Virus Scan' results due to their high observed standard deviation (this high standard deviation was observed for Xen/Ether as well as KVM/Nitro), however the results are present in Table 8.4 for the curious reader. Despite this, we see that Nitro outperforms Ether both on the absolute scores and the amount of degradation in all tests performed. In the benchmarks that focus on arithmetic operations (i. e., use relatively fewer system calls), for example compression and encryption tests, Nitro outperforms Ether only nominally. However in the case of HTML rendering, text editing, and memory latency, Ether's degradation is between 5 and 54 percentage points greater than that of Nitro. As mentioned in Section 8.1, while the PCMark05 benchmarks nicely reveal which specific operations incur the greatest degradation, the PerformanceTest benchmark is a better indicator of the overall degradation of the system. We see that with this benchmark Ether's degradation is over 20 percentage points more than that of Nitro.

We hypothesize that Ether's greater degradation is primarily due to the fact that Ether forces a page fault interrupt to perform system call tracing. This adds additional overhead to a part of the hypervisor which is already responsible for incurring a large performance overhead. Additionally, this design decision effectively counteracts any benefits one might have from using Ether with a hypervisor which makes use of Extended Page Tables.

8.5 Summary

In this chapter we have supported our performance claims with tests. We presented tests across several mainstream client and server OSs and demonstrated that Nitro performs well with all of them. We strengthened this claim by comparing the performance of Nitro with that of Ether [17], another system that performs VMI-based system call trapping, though not in a fully derivative manner. Through this comparison we showed that Nitro's purely derivative approach to system call tracing outperformed Ether in all tests.

Chapter 9

Detecting Malware Using System Calls

Detecting malware is an ever present challenge in the field of security. Traditionally, malware detection makes use of signature-based methods. That is, known malware samples are analyzed to create a repository of signatures which are then matched against a static object to determine whether the particular object is infected with malware. While this approach is straightforward, it has two fundamental issues. The first stems from the static nature of the analysis. A static analysis denotes that the analysis is performed on an inert object, that is, an object that is not being executed or in any other way active. Malware authors take advantage of this fact by obfuscating the inert object in such a way that it no longer matches any of the the signatures in the repository. However, when executed, the actions of the active process prove malicious. This may be achieved by simple packing and unpacking of the malicious portions of the object or by more advanced polymorphism techniques.

The second issue with such an approach is a result of its reliance on signatures. These signatures must be generated prior to a successful match, which makes such an approach disadvantageous in situations where no prior sample existed for signature generation. That is, novel malware that makes use of, so called, "0-day" exploits (exploits which have not yet been seen in the wild) are impossible to detect with a signature-based method. To address these issues, dynamic machine learning-based analysis has often been considered in various forms [65, 41, 64, 92, 72].

A dynamic analysis means the behavior of the malware is analyzed rather than the inert object. This evades traditional code obfuscation as the behavior remains malicious and it is this behavior that is analyzed. Obfuscating behavior becomes much more difficult as the malicious act must be carried out in some form. That is, one can attempt to conceal their intentions, but once the malicious act is carried out, this behavior is ideally observable and can be acted upon. Furthermore, machine learning techniques lend themselves well to malware detection as such techniques make an attempt to generalize and learn the features of malware that differentiate them from benign software. This can then also be applied to novel malware which has yet to be seen in the past.

While such approaches show much promise they are not immune to short-comings of their own. While obfuscating behavior is more difficult than obfuscating code, it is not impossible. Depending on how the behavior of a process is modeled, dynamic analysis is generally vulnerable to a class of attacks called mimicry attacks [83, 84]. This class of attack attempts to "act benign" while secretly carrying out some malicious action. Additionally, machine learning techniques often require the behavior to be modeled in full before classification can take place, making online classification impossible. In addition, the large time complexity combined with the massive amount of data that needs to be classified often makes a practical solution difficult.

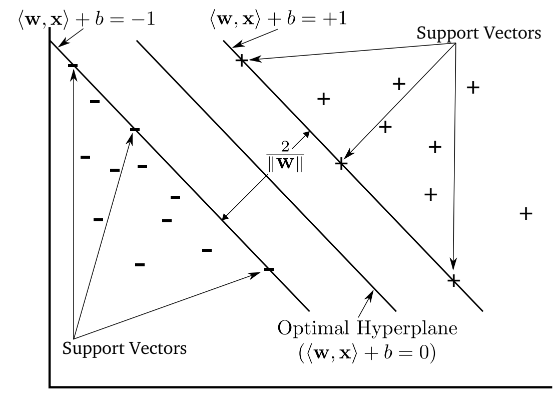
In this chapter, we model process behavior though system call traces collected by Nitro and present a practical machine learning-based method for malware detection. Specifically, we make use of a support vector machine (SVM). Through proper election of a string kernel function and a novel approach to applying the SVM over a sliding window, we address the issues often associated with dynamic machine learning-based approaches and present a technique which we show to be both accurate and practical. With this method, we use system call traces to classify processes in an online manner and are able to achieve an accuracy of 99%. Finally, we argue that due to the flexible nature of our approach along with our choice of kernel function this method raises the bar against mimicry attacks.

9.1 Method

As mentioned in Section 7.1, we choose to focus on system calls due to their significance as the communication channel between kernel and user space. Kolter and Maloof propose modeling process behavior with byte code traces [41]. While this approach is promising as they show and the behavior is modeled very thoroughly through byte code traces, it requires the collection and processing of a lot of data. Furthermore, a lot of this data may be extraneous as the model is a very low-level representation of the process behavior.

In an attempt to reduce the amount of extraneous information, we concentrate on what it is that malicious code will try to achieve. A process in complete isolation cannot perform any malicious action on the rest of the system. Hence, in order for a process to act maliciously it must interact in some manner with the rest of the system and if the isolation mechanism in place is sound, this interaction must take place through the interface provided by the OS (i. e., system calls).

As a result of the above observation, system call traces or API call traces are often used to model process behavior [65, 64, 92]. API call traces are similar in nature to system call traces, though the APIs are implemented at a level above system calls and generally lack hardware assistance in their implementation. It is reasonable to assume that if an approach is successful for either system call traces or API call traces, it will also be successful for the other. We tested our approach on both types of traces, though



$\langle \mathbf{w},\mathbf{x}\rangle + b = -1$
$\langle \mathbf{w},\mathbf{x}\rangle + b = +1$
Support Vectors
$\frac{2}{\|\mathbf{w}\|}$
Support Vectors
Optimal Hyperplane
$(\langle \mathbf{w},\mathbf{x}\rangle + b = 0)$

Figure 9.1: Visualization of the maximization problem used to determine the optimal hyperplane for a support vector machine.

we focus on the system call traces provided by Nitro to model the behavior of a process. Specifically, we will model the behavior of a process with a sequence of system call numbers. System call numbers simply represent a specific OS services and are assigned by the OS. It is also important to note that we make use of Nitro's tracing feature (i. e., we forgo the use of system call arguments). While this information may be useful for certain approaches, we show that our approach is successful without arguments. This also significantly reduces the collection overhead.

9.2 Classification

For the classification of the system call traces, we make use of support vector machines (SVMs) [71]. SVMs are a maximal margin hyperplane classifiers. That is, given a training set $X = \{(\mathbf{x_m}, y_m)\}_{m=1}^{M}$, where $\mathbf{x_m}$ is a training vector and y_m is the associated class $+1$ or -1, the SVM identifies the hyperplane for which the separation between the most relevant training vectors (i. e., the support vectors) and the hyperplane is maximized, then classifies new vectors based on their relation to this hyperplane. The hyperplane is represented by a weight vector $\mathbf{w} \in \mathbb{R}^D$ and $b \in \mathbb{R}$ and is formally defined for some $C > 0$ in the following optimization problem:

$$\begin{aligned} \underset{\mathbf{w},\xi,b}{\text{minimize}} \quad & \frac{\|\mathbf{w}\|^2}{2} + \frac{C}{M}\sum_{m=1}^{M}\xi_m \\ \text{subject to} \quad & y_m(\langle \mathbf{w}, \mathbf{x}_m\rangle + b) \geq 1 - \xi_m, m = 1, \cdots, M \end{aligned} \tag{9.1}$$

Here the notation $\langle \mathbf{w}, \mathbf{x}_m\rangle$ denotes the dot product of the vectors $\mathbf{w}$ and $\mathbf{x}_m$. This problem states that we must find $\mathbf{w}$, b, and a slack variable, ξ such that $\langle \mathbf{w}, \mathbf{x}\rangle + b \geq +1 - \xi$ if $\mathbf{x}$ is in the "+1" class, $\langle \mathbf{w}, \mathbf{x}\rangle + b \leq -1 + \xi$ if $\mathbf{x}$ is in the "−1" class, the margin (i.e. $\frac{2}{\|\mathbf{w}\|}$) is maximized, and the mean of the slack variables (ξ_i) is minimized. The slack variables prevent the SVM from over fitting the model by allowing the left-hand side of the inequality to be slightly less than +1 even if $\mathbf{x}$ belongs to the "+1" class or slightly greater than −1 even if $\mathbf{x}$ belongs to the "−1" class. A visualization of this is depicted in Figure 9.1.

By introducing a Lagrangian with multipliers $\alpha_m \geq 0$, the training phase determines which training vectors will become support vectors. Essentially the training phase will choose a training vector $\mathbf{x_m}$ if the corresponding $\alpha_m > 0$, that is, if the training vector lies directly on the +1 or −1 hyperplane.

Finally, the classification occurs by comparing the test vector to each support vector and measuring the similarity between each support vector and the test vector. The decision function f is formally defined as:

$$f(\mathbf{x}) = \operatorname{sgn}(g(\mathbf{x})) \tag{9.2}$$

where

$$g(\mathbf{x}) = \sum_{m=1}^{M} y_m \alpha_m \langle \mathbf{x}, \mathbf{x}_m \rangle + b \tag{9.3}$$

Here the dot product plays the role of the kernel function, which measures the similarity between the two vectors. For simple geometric classification, a dot product may suffice as a measure of similarity. However, for detecting malware through system call traces, a more complex kernel function is necessary. This kernel function must be carefully chosen for a given domain and is discussed in further detail in Section 9.3.

9.2.1 Probability Estimates

As shown in the previous section, SVMs predict a class without producing a probability associated with this prediction. That is, the output of the SVM is binary, indicating the predicted class as is seen in (9.2). However, it is often beneficial to work with a posterior probability $P(y = 1|g(x))$ based on $g(x)$ defined in (9.3). Such a posterior probability is especially helpful when the output is to be combined with other factors to reach a final decision.

This probability must be estimated and several methods for this estimation have been proposed. We make use of a method proposed by Platt [58]. Platt's method estimates the posterior probability by using the following sigmoid function:

$$P(y = 1|g(x)) = \frac{1}{1 + e^{Ag(x)+B}} \tag{9.4}$$

where A and B are found by minimizing the negative log likelihood of the training data.

9.3 Kernel Function

In looking for a kernel function we begin by examining the nature of the input itself. The input consists of a string (i. e., sequence) of system call numbers. For the language processing domain, string kernels were introduced to classify texts or strings [48]. In essence, our input is very similar, though instead of classifying strings over the roman alphabet, for example, we are interested in classifying strings over the alphabet of all system calls. That is, we define our alphabet, Σ, as all possible system calls and a string is a sequence $s \in \Sigma^*$ of letters (i. e., system calls). Based on this similarity, we choose a string kernel for our method.

Specifically, we choose to use the string subsequence kernel (SSK) [48]. This kernel measures the similarity between inputs by considering the number of common *subsequences*. A subsequence allows for letters between elements of the subsequence, though the kernel penalizes the similarity as this number of interior letters increases. For example, the string ABC would clearly match on the string ABC, but it would also match on the string $AaaaBbbbCccc$, though with a lower similarity measure due to the interior aaa and bbb. This property of the kernel is especially attractive as a sequence of system calls may contain interior system calls that might be irrelevant to the malicious nature of the sequence.

The SSK is formally defined as:

$$k(s,t) = \sum_{u \in \Sigma^n} \sum_{\mathbf{i}:u=s[\mathbf{i}]} \sum_{\mathbf{j}:u=t[\mathbf{j}]} \lambda^{l(\mathbf{i})+l(\mathbf{j})} \tag{9.5}$$

where n is the size of the subsequence and $\lambda \in (0,1)$ is the decay factor used to weight the contribution of the match based on the number of interior letters. The notation $u = s[\mathbf{i}]$ denotes that u is a subsequence of s for which there exist indices $\mathbf{i} = (i_1, \ldots, i_{|u|})$, with $1 \leq i_1 < \cdots < i_{|u|} \leq |s|$, such that $u_j = s_{i_j}$, for $j = 1, \ldots, |u|$. Finally, $l(\mathbf{i})$ represents the length of the subsequence including interior letters.

Previous approaches make use of polynomial kernels or other methods that do not fully consider the sequence of the system calls [64, 72, 65]. That is, in the most trivial case, the number of times that system call occurs in the trace is taken into account without considering the order of the system calls. This is most likely due to the fact that string kernels incur a massive time overhead when used with large amounts of data. However, if one can mitigate the increased time overhead, an approach that considers sequential data has the potential to produce very high accuracy rates. Intuitively, considering sequential data is logical. If one where to manually analyze a system call trace, one would consider the order of the system calls in addition to which system call is being executed. In an effort to baseline the time overhead, we began training the SSK with our raw data and broke the test off after two months of running with no result in sight. So with practical analysis as a goal, clearly this time overhead must be addressed.

9.4 Practical Application

We address the practical application of the SSK with the observation that if we are able to classify a process by updating an interim classification value and making a decision before the process has finished, we inherently address online classification while reducing the time overhead by not having to analyze the entire system call trace.

To prepare the training data, we iterate over each individual system call trace and extract contiguous sub-traces of size S starting at random points within the traces. We iterate over all the training traces several times in order to get several sub-traces from each original trace. These size S sub-traces become our training set. We do this for the sake of practicality because training the SSK with circa 2000 traces, some of which may contain hundreds of thousands of system calls, takes months even on modern hardware. In an effort to make use of the strength of this kernel in a practical manner, we took random size S samples from the original training traces. The clear concern is that some of these sub-traces may not be indicative of the class they belong to because they represent a relatively small fraction of the entire trace. However, with enough sub-traces we will statistically collect some that are indicative of the class they belong to. The beauty of a SVM is that it will decide which of the sub-traces to use as support vectors (hopefully those indicative of the training class) and which to disregard. With enough random sub-traces, the SVM will be able to decide which of these it should retain in the form of support vectors. This training set is then used to train the SVM.

The classification, then, works by sliding a window of size S over the system call trace that is to be classified. This sliding window moves forward by $S/2$ elements in the trace for each iteration. Then, for each iteration, probability estimates are taken using Platt's method [58] as described in Section 9.2.1 and factored into a cumulative moving average for each class. If we let $p_i = P(y = 1|x_i)$ represent the probability estimate as approximated by (9.4) for an iteration i, we represent the cumulative moving average after iteration i as:

$$U_i = \frac{p_1 + \cdots + p_i}{i} \tag{9.6}$$

In addition to calculating the cumulative moving average, we also experimented with a simple moving average of the probability estimates. This is a similar method, though instead of considering all previous window iterations, a simple moving average only considers the last y window iterations in the average (where y may be arbitrarily set). Formally,

$$S_i = \frac{p_{i-y} + \cdots + p_i}{y} \tag{9.7}$$

where $i \geq y$.

We continue our classification by defining two thresholds $T_1 \in [0.5, 1]$ and $T_{-1} \in [0.5, 1]$. These thresholds get compared to U_i and $1 - U_i$, respectively and if either threshold is exceeded, the classification ends by predicting the class represented by the exceeded threshold. Formally, the decision function is represented as follows:

$$D_i = \begin{cases} 1 & \text{if } U_i > T_1, \\ -1 & \text{if } 1 - U_i > T_{-1}, \\ D_{i+1} & \text{else} \end{cases} \tag{9.8}$$

Clearly, $U_i > 0.5 \wedge (1 - U_i) > 0.5$ can never be true if $U_i \in [0, 1]$, therefore if $T_1 \in [0.5, 1]$ and $T_{-1} \in [0.5, 1]$, only one single case of the decision function will ever be true for a given iteration. For practicality, if the cumulative moving average never exceeds either threshold and there are no more system calls in the trace, the decision function simply predicts 1 if $U_i > 0.5$ or it predicts -1 otherwise.

9.5 Summary

This chapter proposes a novel method for practical, online malware detection using machine learning methods. We make use of a SVM in combination with the SSK. This string kernel was originally proposed for text classification, but has properties which make it attractive for the malware detection domain. We address the large time overhead generally associated with such an approach by considering the moving average of probability estimates over a sliding window that is slid over system call traces. This moving average is then compared to a threshold to predict a class. Using a machine learning approach sets our method apart from the more traditional signature-based methods in that such methods have been shown to work for novel malware, that is, malware which has yet to be seen.

In the following chapter, we present an evaluation of our approach. Our experimentation will show that this method is both accurate and considerably reduces the time overhead associated with using the SSK for this domain.

Chapter 10

Malware Detection Evaluation

In this chapter we present the results of our experiments when testing Nitro and our SVM-based method for malware detection on real-world data. Additionally, we take special care to discuss the proper setup and preparation of the data to be trained on and tested.

10.1 Data Collection

We ran this experiment on two sets of sample traces. Both sets of samples include both malicious and benign samples that were run on the Windows XP operating system (OS). We chose Windows XP as it is a popular commercial OS and numerous malware samples are available for this platform.

The first set of system call traces was collected using Nitro. This dataset includes 1943 system call traces of malicious samples taken from VX Heavens [1] and 285 system call traces of benign samples taken from a default Windows XP installation and selected installations of well-known, trusted applications.

The second set of traces works on a level slightly above system calls. Windows XP wraps its system calls in APIs that it provides to programmers through system libraries. While these traces are technically at a level slightly above the system calls themselves, they serve the same purpose and demonstrate that our method works at both levels. This dataset is collected by hooking these API functions and was first used by Xiao and Stibor [92]. It consists of 2176 API call traces of malicious samples and 161 API call traces of benign samples.

We chose to introduce the second independent data set for two reasons. First, the second data set makes use of API call traces rather than system call traces directly. This gives us a chance to observe the accuracy of our approach for system calls as well as API calls. The second and perhaps more important reason for including a second

[1] `http://www.vxheavens.com`

data set is to confirm that our method also works on an independent data set that was not collected by us. This strengthens the credibility of our approach as it allows us to present results based on data that others have previously used in similar experiments. In fact, we directly compare the results of our method with that of Xiao and Stibor in Section 10.4.

10.1.1 Imbalanced Data

One may notice that there are significantly fewer benign traces than malicious traces in both data sets. This can lead to misleading results if experiments are not conducted correctly. The first step in addressing imbalanced data is to use a technique called *cross validation*. This technique ensures that all data is used for both training and testing, but in a manner such that in a single iteration the training data and testing data remain completely independent. Allowing a classifier to train on data, then testing on that same data will result in biased results. It is not difficult to create a classifier that is able to classify data that it has seen before in a training phase. The simplest of such an approach is a signature-based approach. However, we want to show that our approach also works for previously unseen malware. To address this, cross validation involves partitioning the data and training on one labeled partition then classifying the remaining partitions without a label. Once this is complete, the process is repeated with a new partition as the training set. This ensures that the classifier is always receiving data it has never seen before during the testing phase. Specifically, how we set this up in our evaluation is described in detail in Section 10.2

The second precaution that one must take to avoid misleading results is to consider both the ratio of false positives to total negatives (the false positive rate) and the ratio of true positives to total positives (recall) in addition to the precision and accuracy of one's approach. When training classifiers with data sets in which the classes are significantly imbalanced (as is the case in our data set), there is a risk that the classifier "learns" this imbalance and bases its classification, in part, on the distribution of the labels in the training set. This is logical, if one is unsure how to classify something, the best guess might be to rely on the distribution of previously observed data points. However, this can lead to misleading results. For example, consider a testing set in which the number of positives (P) is 99 and the number of negatives (N) is 1. Assume that the classifier classifies all 100 data points as P. We might conclude that our classifier classified the testing data with an accuracy of 99% (accuracy $= \frac{TP+TN}{P+N}$, where TP = number of true positives and TN = number of true negatives). However, this is misleading as the classifier classified *none* of the elements in the N class correctly. In this case, a false positive rate of 100% reveals this fact (false positive rate $= \frac{FP}{N}$). This illustrates the fact that in addition to accuracy and precision, one should take care to consider the false positive rate and the recall to identify misleading results. Looking at these measures in addition to accuracy and precision will give a clearer picture of the success of one's approach.

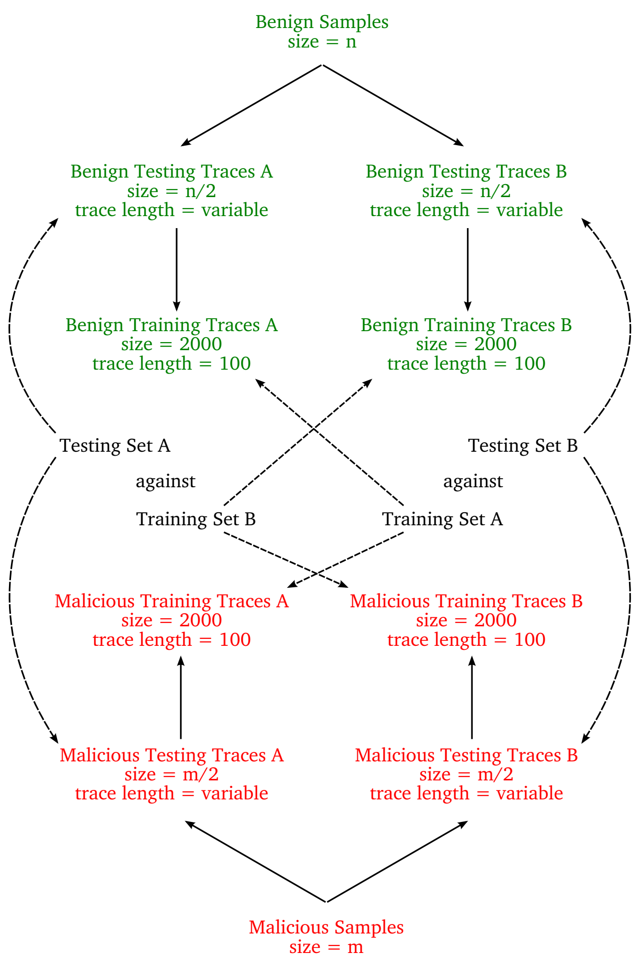


Figure 10.1: Splitting the sample sets for a two-fold cross validation.

10.2 Setup

We begin by preparing the training set. Since the training and testing sets must be prepared differently as described in Section 9.4 and we need to set up a cross validation, this requires a few extra steps. We begin by splitting both the set of malicious traces and benign traces in half. These sets can be directly used in testing as this step shifts its sliding window over the entire trace. However, to produce the training set, we iterate over each testing set and extract random contiguous subsequence of size S. In our experiments, $S = 100$ and we iterate 2000 times. This results in four testing sets of traces (two malicious and two benign), each with training set of 2000 random contiguous subsequences created from it. We then combine the sets, such that a single malicious testing set is combined with a single benign testing set as well as the respective training sets. With the training set prepared, we can now perform a standard two-fold cross validation in which the traces used for the training set are *completely independent* of the traces used for the testing set. The creation of these sets is depicted in Figure 10.1.

With the data collected and prepared, we make use of LIBSVM [13] along with a provided string kernel extension to perform both training and classification as described in Section 9.4. Since the SSK kernel is not implemented in LIBSVM or the string kernel extension, we incorporated the SSK implementation proposed by Herbrich [33]. These implementations had to be further modified such that they accept an input over an integer alphabet as opposed to a roman letter alphabet used in text classification. It is also important to note that LIBSVM calculates probability estimates by making use of an improved algorithm for minimizing the negative log likelihood proposed by Lin et al.[45].

With these tools and the data prepared, we set up the experiment as described in Section 9.4. We also found that it was necessary to factor several values into the moving average before checking either threshold. This allows the moving average to factor in some data before a decision is made. For this reason we always factor the probability estimates for the first 10 iterations in our moving average before we begin considering the thresholds.

10.3 Results

For each experiment, P represents the number of positive (malicious) samples and N represents the number of negative (benign) samples. Then, as each experiment runs, we collect the number of correctly and incorrectly classified results as true positive (TP), true negative (TN), false positive (FP), and false negative (FN). With this information we calculate the classification measures presented in Table 10.1 and Table 10.2 and discussed below. Finally, we also present the number of average iterations (the average number of times the SVM classifier had to be called until either threshold was met) and the moving average type (CMA = cumulative moving average, SMA = simple moving average) for each experiment.

We began by considering the threshold values. These values are quite important as they most directly affect average number of iterations it takes to make a decision. As the thresholds rise, so does the average number of iterations in general. However, if a threshold is too low the number of false positives and/or negatives rises. In addition, we noticed that our SVM was much more sensitive to malicious samples than it was to benign samples. That is, the average of the malicious probability estimates rose much more quickly to 1 for malicious samples than the average of the benign probability estimates for benign samples. This observation led us to set the benign threshold higher than the malicious threshold.

We next considered the variable n. As mentioned in Section 9.3, n is the size of the subsequence that the kernel function looks for in the traces being compared. What we observed is that as we increased n from the initial value of 3, the classification measures for both datasets became worse. This may seem somewhat counterintuitive. However,

Test Num.	n	λ	T_{-1}	T_1	Avg.	TP	FP	TN	FN	Average Iterations
1	**3**	**0.5**	**0.5**	**0.75**	**CMA**	**1929**	**11**	**274**	**14**	**13.2608**
2	4	0.5	0.5	0.75	CMA	1910	15	270	33	29.4753
3	5	0.5	0.5	0.75	CMA	1903	20	265	40	19.7244
4	3	0.25	0.5	0.75	CMA	1702	17	268	241	209.4165
5	3	0.75	0.5	0.75	CMA	1902	14	271	41	120.6194
6	3	0.4	0.5	0.75	CMA	1919	12	273	24	14.1831
7	3	0.6	0.5	0.75	CMA	1929	14	271	14	22.4475
8	3	0.5	0.5	0.5	CMA	1941	35	250	2	10.8406
9	3	0.5	0.5	0.75	SMA	1869	7	278	74	11.3321
10	3	0.5	1	0.9	SMA	1906	48	237	37	722.3012

Test Num.	FP Rate $\frac{FP}{N}$	Recall $\frac{TP}{P}$	Precision $\frac{TP}{TP+FP}$	Accuracy $\frac{TP+TN}{P+N}$	F-Measure $\frac{2}{\frac{1}{\text{Precision}}+\frac{1}{\text{Recall}}}$
1	**0.0386**	**0.9928**	**0.9943**	**0.9888**	**0.9936**
2	0.0526	0.9830	0.9922	0.9785	0.9876
3	0.0702	0.9794	0.9896	0.9731	0.9845
4	0.0596	0.8760	0.9901	0.8842	0.9295
5	0.0491	0.9789	0.9927	0.9753	0.9857
6	0.0421	0.9876	0.9938	0.9838	0.9907
7	0.0491	0.9928	0.9928	0.9874	0.9928
8	0.1228	0.9990	0.9823	0.9834	0.9906
9	0.0246	0.9619	0.9963	0.9636	0.9788
10	0.1684	0.9810	0.9754	0.9618	0.9782

Table 10.1: Experimental results for *system call trace dataset* ($P = 1943$, $N = 285$) The top table displays the set-up parameters and raw results while the bottom table represents the classification measures.

Test Num.	n	λ	T_{-1}	T_1	Avg.	TP	FP	TN	FN	Average Iterations
1	**3**	**0.6**	**0.5**	**0.75**	**CMA**	**2029**	**19**	**142**	**147**	**37.2174**
2	3	0.5	0.5	0.75	CMA	1971	19	142	205	31.3479
3	3	0.4	0.5	0.75	CMA	1699	15	146	477	32.8015
4	4	0.5	0.5	0.75	CMA	1660	20	141	516	29.1656
5	5	0.5	0.5	0.75	CMA	1697	18	143	479	32.2657
6	3	0.5	0.5	0.75	SMA	1456	16	145	720	14.1694
7	3	0.5	0.7	0.9	SMA	1961	21	140	215	51.3697

Test Num.	FP Rate $\frac{FP}{N}$	Recall $\frac{TP}{P}$	Precision $\frac{TP}{TP+FP}$	Accuracy $\frac{TP+TN}{P+N}$	F-Measure $\frac{2}{\frac{1}{\text{Precision}}+\frac{1}{\text{Recall}}}$
1	**0.1180**	**0.9324**	**0.9907**	**0.9290**	**0.9607**
2	0.1180	0.9057	0.9905	0.9042	0.9462
3	0.0932	0.7808	0.9912	0.7895	0.8735
4	0.1242	0.7629	0.9881	0.7706	0.8610
5	0.1118	0.7799	0.9895	0.7873	0.8723
6	0.0993	0.6691	0.9891	0.6851	0.7982
7	0.1304	0.9012	0.9894	0.8990	0.9432

Table 10.2: Experiment results for *API call trace dataset* ($P = 2176$, $N = 161$) The top table displays the set-up parameters and raw results while the bottom table represents the classification measures.

n is very dependent on S (the size of the window). Since the SSK function does not compute distance between matched subsequences it must look for exact subsequence matches and as n approaches S the probability that two size n subsequences exist in two separate traces of relatively small size decreases. This may become clear through a simplified example. Let us assume that we are matching *substrings* of size 3 (i. e., $n = 3$) in two strings of size 3 (i. e., $S = 3$) and our kernel function simply returns the number of matches. If we consider the two stings to be matched to be "aab" and "aac", our kernel function returns 0 (i. e., no match). However, if we set $n = 2$, our kernel function returns 1, indicating some level of similarity, for the same input.

We then began to experiment with various values for λ. $\lambda \in (0, 1)$ is the decay factor used to weight the contribution of the match based on the number of interior letters. That is, as λ approaches 1, interior letters are increasingly penalized. We were surprised by the drastic increase in the average number of iterations it took for a decision to be reached as λ moved away from 0.5. This can most dramatically be seen for values $\lambda = 0.25$ and $\lambda = 0.75$ in Table 10.1. We found that these values caused the probability estimates to remain closer to 0.5, this caused the decision function to take longer when the probability estimates where favoring the malicious (i. e., "+1") class, as this threshold is set to 0.75. We concluded that a value close to 0.5 was best for both datasets.

Finally, we considered using simple moving averages as opposed to cumulative moving averages to make a classification. We found that using a cumulative moving average performed slightly better than a simple moving average. We reasoned that because the average number of iterations is so low when using the cumulative moving average, the success of two methods would not differ greatly if all other factors remained the same. One would expect to see a greater difference in the performance of the two methods if the average number of iterations is much higher. This is supported by the API call dataset in which the average number of iterations is higher and the success of the two methods differ more greatly. In both cases, however, the experiments that made use of a cumulative moving average performed better.

After having experimentally optimized the various variables, we see that $n = 3$, $\lambda = 0.5$, $T_{-1} = 0.5$, $T_1 = 0.75$, and using a cumulative moving average produces the best results for the system call datasets. We show that these values contribute to a 99.28% recall, a 99.43% precision, a 98.88% accuracy, and a 99.36% F-measure, with only a 3.86% false positive rate. We performed more thorough testing on the system call data set as it is the data we collected and it is the system call traces that our system focuses on rather than API call traces. We tested our method on the second dataset (i. e., the API call dataset) to strengthen our claim that our approach performs well. For this dataset, we produced the best results with $n = 3$, $\lambda = 0.6$, $T_{-1} = 0.5$, $T_1 = 0.75$. With these inputs, our approach produced a 93.24% recall, a 99.07% precision, a 92.90% accuracy, and a 96.07% F-measure, with a 11.80% false positive rate.

10.4 Comparison

In this section, we compare the results of our approach with those of other approaches. To our knowledge there are no other approaches that make use of string kernels with SVMs, however we compare our approach with SVM-based approaches, an approach that makes use of a k-nearest neighbor (kNN) classifier, and an approach that makes use of probabilistic topic models.

10.4.1 SVM-based Approaches

We compare our approach to two SVM-based approaches that make use of differing kernels.

Author	Approach	FP Rate	Recall	Accuracy	
Pfoh	SVM+SSK (syscalls)	0.0386	0.9928	0.9888	(1)
Pfoh	SVM+SSK (API)	0.1180	0.9324	0.9290	(2)
Rieck et al. [64]	SVM+Poly	☐	☐	0.88	(3)
Rieck et al. [64]	SVM+Poly (extended)	☐	☐	0.76	(4)
Wang et al. [85]	SVM+GRBF	0.0222	0.995	☐	(5)
Liao and Vemuri [44]	kNN (total)	0.0	0.917	☐	(6)
Liao and Vemuri [44]	kNN (novel)	0.0	0.75	☐	(7)
Xiao and Stibor [92]	STT	0.4286	0.9955	0.9721	(8)
Xiao and Stibor [92]	STT+SVM	0.3748	0.9997	0.9790	(9)

Table 10.3: A comparison of results from various machine learning approaches to malware detection using system call traces. The ☐ symbol indicates that the information is not available.

10.4.1.1 Polynomial Kernel Function

The first approach we will compare our results with is the work of Rieck et al. [64]. This approach models the system trace by counting the frequency of each system call. The frequency of a system call becomes the weight of that particular system call and this information is stored in a separate vector for each trace. These vectors can then be introduced as arguments to a kernel function. In this case, Rieck et al. make use of polynomial kernel:

$$k(\mathbf{x}, \mathbf{y}) = (\langle \mathbf{x}, \mathbf{y} \rangle + 1)^d \tag{10.1}$$

where d is the dimension of the vector.

For their testing, they made use of a corpus of 10,072 malware samples divided into 14 malware families. Of interest is that the authors classified the malware into the 14 different classes rather than classifying a process as either malicious or benign. This level of granularity is helpful especially when trying to determine whether a specific malware sample is similar to a family of malware that is already known or whether it represents a potentially new family of malware. However, such an approach also requires a classifier for each family which can become very cumbersome as the number of malware families grows. Despite this, the authors also published the overall accuracy which allows us to nicely compare our results with theirs.

The results of this approach can be seen in Table 10.3, lines 3 and 4. Line 3 represents a round of testing the authors did using normal cross validation as is the case in our testing, while Line 4 represents testing that took place with an extended dataset that included malware that belonged to none of the malware families along with benign processes. We see that, comparatively, our approach is more accurate. This is not surprising as the approach used by Rieck et al. does not consider any sequential information at all.

10.4.1.2 Gaussian Radial Basis Function

The approach published by Wang et al. [85] also makes use of SVMs although they make use of n-grams. That is, this approach measures the occurrence of n-grams rather than individual system calls. The strength of an approach that makes use of n-grams is that some sequential data is preserved. This is due to the fact that an n-gram is a sequence of system calls and within one n-gram the sequential information is preserved. A single system call trace is then examined and the presence of n-grams is noted. That is, each n-gram is represented by a position in a vector and the value at each position is 1 if the respective n-gram exists in the trace and is 0 otherwise. Such a vector is built for each trace and delivered to a Gaussian Radial Basis Function (GRBF):

$$k(\mathbf{x}, \mathbf{y}) = e^{-\gamma\|\mathbf{x}-\mathbf{y}\|^2} \tag{10.2}$$

where γ is "optimized through experiments and comparisons" [85].

The authors made use of 722 benign executables and 1589 worms in their experimentation and one can see by looking at Line 5 of Table 10.3, that their approach is quite successful. In fact, it seemingly outperforms our approach by a small margin. However, one must be very careful to notice that the authors only classify benign executables and *worms*. That is, this approach only considers one family of malware. This has two ramifications. First, it is useless when classifying other classes of potentially harmful malware. Second, while the metrics presented in Table 10.3 may be close to those of our approach, one cannot conclude that it is a better approach when applied to *all* malware families. This is due to the fact that as one restricts a class, its members share more similarities making it easier for a classifier to correctly classify an unknown. For example, a classifier that is tasked with classifying humans into classes, **males** and **females**, might have a more difficult time than a classifier who is tasked with classifying humans into classes, **brunette males** and **females** (non-brunette males have effectively been removed from our world). This is due to the fact that if the observed human is not brunette, the classifier can already classify it as a female based on this information alone.

As opposed to the approach from Rieck et al. [64], Wang et al. [85] make use of n-grams to preserve some sequential information, however with their approach they loose quite of a bit of frequency information as they simply consider the existence of an n-gram rather than the number of occurrences. Additionally, they loose potentially valuable information in that they only run each executable for 10 seconds. This makes such an approach very susceptible to mimicry attacks. This class of attacks is further elaborated in Section 10.6.

10.4.2 k-nearest Neighbor Classifier

Liao and Vermuri [44] present an approach that makes use a k-nearest neighbor (kNN) classifier. This approach is described in Section 7.1.1, however we will briefly recall for the readers benefit. A kNN classifier makes use of frequencies by storing the frequency of a single system call on a per trace basis. That is, to train such a classifier each trace is processed and the frequency with which each system call is used is stored per trace. In order to classify an unknown trace, the classifier computes the k most similar traces from the training set and classifies the unknown trace based on the labels associated with the k most similar traces. In this instance the authors make use of the cosine similarity:

$$s(\mathbf{X}, \mathbf{Y}_j) = \frac{\sum\limits_{i=1}^{n} x_i y_{j,i}}{\|\mathbf{X}\| \|\mathbf{Y}_j\|} \tag{10.3}$$

where $\mathbf{X}$ is the unknown trace, $\mathbf{Y}_j$ is the j^{th} training trace, n is the number of system calls, x_i is the frequency of the i^{th} system call in $\mathbf{X}$, and $y_{j,i}$ is the frequency of the i^{th} system call in $\mathbf{Y}_j$.

For their experimentation, the authors made use of 5285 benign traces and 24 malicious traces. When training they used 16 of the 24 malicious traces. This leads to a situation in which the results in Line 6 of Table 10.3 include the same 16 of 24 traces when testing as when training. Clearly, the classifier classified these 16 traces 100% correctly. Therefore, the results on Line 7 of Table 10.3 represent results that are a better measure of the approach. Despite this, our approach achieves a higher recall than both approaches and the 0% false positive rate for each test can be attributed to the fact that there are far more benign traces than malicious traces.

10.4.3 Probabilistic Topic Model

Finally, Xiao and Stibor present an interesting approach that makes use of the supervised topic transition (STT) model described in Section 7.1.1. This approach assigns system calls to topics. That is, the algorithm groups the system calls based on co-occurrence. The model is then built by modeling the the topic transitions rather than the system call transitions that one might expect.

This approach makes use of an algorithm that iteratively alternates between a Gibbs sampling approach and a gradient descent approach to update the topic assignment and the topic transition model in parallel to train the algorithm. The classification, then, takes place by generating a topic transition model for the unknown trace and probabilistically predicting a label.

In addition to a pure STT approach, the authors also considered a classifier that makes use of a SVM. In this instance, the same training method is used, however the topic transitions are fed into a SVM. This SVM makes use of a RBF kernel similar to the kernel described in Section 10.4.1.2.

In their experimentation, the authors made use of 2880 malicious traces and 168 benign traces and tested several methods. The two most successful are described here and the results are depicted in Table 10.3. Line 8 represents the pure STT approach while Line 9 represents the approach in which the authors combined their STT model with a SVM classifier. While this approach performs slightly better than our approach when considering the recall, the fact that they report a 37% and 43% false positive rate favors our approach in this regard.

10.5 Online Classification

As mentioned in Section 9.4, our method inherently lends itself to online classification due to the fact that it considers additional system calls as they are produced by the process. However, we must also consider the time overhead. The issue with classifying an entire system call trace using the SSK is that a single trace may be hundreds of thousands of system calls long and examining two traces of this length for matching subsequences will clearly lead to a large time overhead. We solve this problem by keeping the lengths (S) of the traces that we input into the SVM relatively small (100 system calls).

By setting S to a relatively small value we make using the SSK feasible. However, in order for our classification to be accurate, we need to iterate over some number of windows before a final decision can be made. That is, we must still consider the number of iterations that it takes our method to make a decision. As is shown in Table 10.1, the average number of iterations for the experiment with the highest accuracy is 13.26, while in Table 10.2 the average number of iterations for the experiment with the highest accuracy is 37.22. That is, our method of classification can make a decision after only considering a relatively small number of system calls, which significantly reduces the time overhead and allows for online classification.

One may criticize the point that our approach does not consider the entire trace, however all such approaches must address this practical problem somehow. The problem is that one may have to wait an indefinite amount of time for a process to finish. For example, a permanently resident process will only end execution once the system is shut down. That is, practically, one will always have to set a maximum trace length to address this and other approaches do this arbitrarily [85] while our approach makes use of the given thresholds to determine when to stop.

Finally, our approach lends itself to online classification due to the fact that the classification can be completely decoupled from the collection. In this case, the only overhead on the monitored system is the collection overhead which remains minimal. The performance degradation when using Nitro to collect all system calls across the entire system is 14%, leaving the user with a very usable system.

10.6 Mitigating Mimicry Attacks

Mimicry attacks [83, 84] are a class of attacks in which either an adversary drowns the individual steps necessary for delivering the malicious payload in “benign steps” or an adversary “acts benign” for a certain amount of time before delivering a malicious payload.

While this class of attacks is certainly a concern for any system that models program behavior through system call traces, the use of the SSK significantly raises the bar against this type of attack. As mentioned in Section 9.3, the SSK matches on subsequences, where the definition of a subsequence allows for interior system calls. In a simple case, if we consider the system call sequence “12,19,39” to be indicative of malicious behavior, a mimicry attack might try to fool the security mechanism by introducing interior “benign” system calls. For example, the attacker might augment the malicious program such that the system call trace was as follows: “**12**,17,13,**19**,32,**39**”. This may be enough to fool signature-based or simple “bag of words” approaches to malware detection, but the beauty of the SSK is that it, by design, will still match on these traces.

On the other hand, if an adversary decides to “act benign” for a time before delivering a payload, our approach may miss the payload if either threshold has been met. The solution for this is simply to raise the threshold. Since the collection can be completely decoupled from the classification as mentioned in Section 10.5 and Nitro introduces a mere 14% performance degradation when collecting all system calls across the entire system, this raising of the threshold only puts strain on our classification and not necessarily on the monitored system. In the most extreme case, one could raise the threshold for the benign class to 1.0. This will result in a system that continuously scans a trace and will only exit if a malicious classification is made. Such an approach would also be applicable in detecting injected code (e. g., shellcode). Due to the fact that any process will be scanned until the threshold for malicious activity is reached, any benign process that is injected with malicious code will also be potentially detected. In order for such an approach to be successful one would most likely have to consider the simple moving average of the probability estimates as described in Section 9.4. We performed such a test on our system call data and were able to produce 96% accuracy as can be seen in Table 10.1.

10.7 Summary

Our experimentation shows that this method is both accurate and considerably reduces the time overhead associated with using the SSK for this domain. We test our method on two separate datasets, collected by differing methods from different levels within the OS architecture. The fact that our method shows promising results for both datasets makes us confident that this method is applicable for other platforms which make use of system calls or API calls as well. Additionally, we compared our approach with other machine learning-based approaches and could show that our approach performs very well in comparison.

Using a machine learning approach sets our method apart from the more traditional signature-based methods in that such methods have been shown to work for novel malware, that is, malware which has yet to be seen. This is supported in our experimentation in that we use a two-fold cross validation. Finally, we argue that our approach raises the bar against mimicry attacks through the use of the SSK and our threshold mechanism.

Chapter 11

Conclusion

Virtual machine introspection is a technique that lends itself well to security applications and as the ubiquity and adoption of virtualization technology continues to increase, we will see VMI become a standard tool for security researchers and engineers. In addition to traditional applications for VMI, such as malware analysis [12, 17, 56, 57, 63, 61], forensics [12, 56, 61, 20, 32, 53], and intrusion detection or prevention systems [39, 46, 12, 56, 53], the growth and advancement of cloud computing hold promising applications for VMI [6, 50, 15]. Although VMI research has grown since Garfinkel et al. introduced Livewire [26], it is only recently that researchers are beginning to look beyond simply placing in-guest solutions into the hypervisor. We are beginning to see approaches that are really represent "out of the box" thinking, such as systems for automatic semantic gap bridging [69, 70, 18], injecting code into the guest [31, 73] or moving code out of the guest [76] at runtime, and instruction trace reconstruction [82] in addition to pure derivation approaches such as Nitro. We have made a case that VMI is moving in a strong direction academically as well as in industry and there are still very many interesting research directions within this field. One of these directions is derivative VMI methods.

We began by taking a closer look at VMI and the process involved. This began by introducing the notion of states and views, in which a state is the representation of the guest system state in its raw form and a view is a concise representation of that state. The challenge is generating a view from the state which is a challenge that all VMI approaches share and is commonly referred to as the semantic gap [14]. The semantic gap represents the disconnect between data and its meaning. That is, the hypervisor sees the data in its raw binary state and there is a step required to move from this raw state to an understandable view of the state. So, the process of bridging the semantic gap results in a view and once this view has been generated, a VMI component can work on it.

As stated, this view generation (i. e., bridging the semantic gap) is a hurdle that all VMI methods must address. What becomes interesting is how they generate a view. To address this, we present three patterns for view generation namely, out-of-band delivery, in-band delivery, and derivative. Both delivery-based methods rely on semantic information about the software that is delivered either before introspection takes place or during introspection, respectively. On the other hand, derivative methods make use of hardware knowledge to derive information about the guest OS. This method has the drawback that not all information about the guest can be derived from the hardware, however it also has two very interesting advantages. First, such approaches are completely guest OS agnostic. That is, as the approach relies on knowledge of the hardware, the software that is running is irrelevant to the method used to generate a view. Therefore, this approach will work for all guest OSs. Second, when using a derivative approach one can be sure that any assumptions about the hardware that are true will remain true for the remainder of the run time of the system. In contrast, when one relies on knowledge of the software, as is the case in delivery methods, one can *not* be sure that the assumptions made will hold true for the entirety of the introspection. That is, one might assume that a data structure lies at a particular offset, however this might change at run-time either for malicious or benign (e. g. as the result of an update) reasons. In either case, the VMI component may not even realize that such a change has taken place and continue to act as though the data structure in question has not moved. Such a scenario can never happen when using derived information as the hardware specifications will not change at run-time. Having identified derivative VMI methods as an interesting approach, we introduced related work in Chapter 4 and demonstrated that very little work has been done to explore pure derivative VMI methods.

We followed this by looking at feasible goals for VMI and to that end, we considered transparency. Transparency can be divided into two types: VMM transparency and VMI transparency. Garfinkel et al. take the position that VMM transparency is infeasible [24]. Taking their thesis as a basis, we extended this argument to include VMI transparency. That is, we argued that it is infeasible to construct a VMI mechanism that is completely transparent to a dedicated adversary. To make this argument, we generalized the thesis of Garfinkel et al. to state that it is infeasible to make two independent hardware platforms indistinguishable, regardless of whether one or both of those platforms is a virtual hardware platform or not. That is, the same points that one can used to argue that an Intel x86 hardware platform will be distinguishable from a KVM virtual hardware platform can be used to argue that an Intel x86 hardware platform will be distinguishable from an AMD x86 hardware platform or that a KVM virtual hardware platform will be distinguishable from a KVM virtual hardware platform that includes Nitro.

Once we accept that full transparency is not possible, we must look for alternative, feasible goals. To this end, we looked at the primary motivations for transparency and discovered that there are two: avoiding observation and protection against evasion. With regard to avoiding observation, we argued that there is little one can to completely eliminate the observer effect, though we can try to make detection very difficult and VMI is already a giant step in this direction. With regard to protecting one's VMI mechanism against evasion techniques, we argued that transparency is not even necessary and rather defined two properties that protect against evasion attempts, namely: evasion evidence and evasion resistance. Both use the notion of hardware rooting as described in Section 5.3.1 to describe properties in which evasion attempts are detected and blocked, respectively.

Ar this point, we began by looking at the Intel x86 architecture in a general manner to identify portions of the hardware that might lend itself to derivative VMI methods. We split our discoveries into three categories: mechanisms that make use of the virtualization extensions, inspection of VM state, and system interrupt-enabled mechanisms. Mechanisms that make use of the virtualization extensions are most straight forward as these are well documented. However, these mechanisms may not have been designed with VMI in mind so a mechanism designed for another purpose may have to be abused to be applicable for VMI. An example of such an approach is the work done by Vogl et al. [82] in which the authors used the performance monitoring counters, which were initially intended to monitor the performance of a system, to perform instruction tracing. The next category, i. e., inspection of VM state, is a passive method of VMI that requires one understand the purpose of the state from a hardware perspective. That is, if one is aware of the function of a register, for example, one can derive information about the state of the system. In this case, the exercise becomes identifying those portions of the state that will allow one to derive useful information. Some such examples were presented in Section 6.2.1. The final category, namely system interrupt-enabled mechanisms, represents mechanisms for which there is no direct method for trapping a desired event. In such a case, one must investigate to see if there is an opportunity to induce a system interrupt which can be trapped to the hypervisor. For example, one must find a way to manipulate the system such that every time the event one is interested in occurs a general protection fault takes place. Since the hardware extensions allow us to trap a general protection fault, we have effectively found a mechanism for trapping the event we are interested in.

With the various methods for derivative VMI outlined, we introduced a proof-of-concept system, Nitro. Nitro performs system call tracing and monitoring in a fully derivative manner. In Chapter 7, we began by making a case for the usefulness of tracing and monitoring system calls. We argued that, as system calls are the method of communication between processes and the kernel, trapping and analyzing them can be helpful in a variety of security applications. We then presented the implementation details of Nitro and showed that it is capable of trapping system calls independent of the system call mechanism that the guest OS chooses to leverage. This was followed by a discussion of the properties held by our system. We showed that Nitro is fully guest OS agnostic and portable, that it is evasion-resistant, and that, while no VMI method can achieve complete transparency, it remains stealthy enough to be applicable in efforts where the observer effect should be mitigated. To strengthen the argument for Nitro's contribution even further, we presented a number of performance testing results in Chapter 8 to show that the collection overhead incurred is very slight. We tested

Nitro in Windows XP (32-bit), Windows 7 (64-bit), 32-bit Linux, and 64-bit Linux environments all with great success. Finally, we compared our system to a system that similarly performs system call tracing, though not in a fully derivative manner, namely Ether [17]. We were able to show that Nitro outperforms Ether in all tests performed and in some cases by quite a margin.

As Nitro is capable of collecting system call traces, the next logical step is to show that one can use these system call traces to accomplish something useful from a security perspective. To this end, we introduced a SVM-based system for classifying processes based on the system call traces produced by Nitro. This approach makes use of a string kernel usually used in the natural language processing domain and attempts to mitigate the usually high performance overhead associated with such approaches. We achieved this by keeping the input into the SVM relatively small and running a sliding window over the system call trace. This addressed the performance overhead in addition to allowing an online classification. We followed this up with an evaluation with real-world data collected by Nitro and found that we could achieve an accuracy of 99%. In order to further strengthen our claim we also tested our method on a dataset collected by Xiao and Stibor [92] as well and were able to achieve an accuracy of 93% using our method. As a further step, we compared the results of our testing with the results of other published approaches that perform malware detection using system call traces. We compared our approach with several SVM-based approaches, a k-nearest neighbor approach, and an approach that makes use of probabilistic topic models and performed among the best of the approaches. While we were able to present the highest accuracy, some approaches presented marginally better recall and false positive rates. We concluded the evaluation of our method with a discussion about mimicry attacks. Mimicry attacks are a class of attacks in which an adversary tries to drown out malicious actions with benign actions. The strength of our approach against such attacks is a result of the string kernel we use. As with natural language processing in which the "meaning" of a text may be buried under inconsequential content, so too the malicious nature of a system call trace may be buried under benign (i. e., inconsequential) content. Since the string kernel we use attempts to address this for natural language processing, it is also applicable against mimicry attacks.

11.1 Looking Forward

As there is little other work specifically looking at derivative methods for VMI, there remains a good amount to be considered in this field. In Chapter 6, we outlined many possible mechanisms in the x86 architecture that would lend themselves to a derivative approach. This might be used as a starting point for finding further applications of the derivative method for the x86 architecture. Undoubtedly, this is not an exhaustive compilation of possibilities and there are very likely many potential derivative applications for the x86 architecture yet to be explored.

As derivative approaches are tailored to a specific hardware architecture, there are many other hardware architectures one might want to consider in the future. Specifically, as embedded devices become increasingly popular looking to their hardware architectures becomes interesting. As of this writing, ARM has just released a new processor in their popular line of embedded processors with full hardware virtualization support [7]. Such architectures are especially interesting from a security and virtualization perspective in addition to what can be done from a derivative VMI perspective.

Finally, approaches that combine both derivative and delivery methods in a clever manner can be very powerful when crafted correctly. While there has been relatively more work done in this area when compared to strictly derivative methods, combined methods have the potential to integrate the power of both methods.

11.2 Final Words

This work began as a keen interest in virtualization and its impact on IT security. This lead to an exploration of the field as a whole which resulted in the discovery of Garfinkel and Rosenblum's foundational work in VMI [26]. With this work and several others in hand, we continued our exploration and realized a need to be able to classify the various methods and approaches. It is out of this need that we developed and published the patterns for VMI presented in this thesis. As simple as they may seem, this was quite a profound step as it lead to our discovery of the derivative approach to VMI. This lead us to the question that would fundamentally drive our research for the next several years, "What information can one derive about a guest system, given that one has no knowledge about the software that is running?". We found this to be a compelling question and began by looking at the x86 architecture, pouring over Intel's architecture manuals. This culminated in our framework, Nitro, which lead to a, slightly tangential, continuation of our research with a look a system calls from a security perspective. To this end, we presented our SSK SVM-based approach for malware detection using system call traces. This has been a lively, interesting, and informational passage which has lead to several important discoveries.

Bibliography

[1] Linux kernel virtual machine. `http://www.linux-kvm.org`. last access: July 7, 2012.

[2] Norman sandbox. `http://www.norman.com/about_norman/technology/norman_sandbox/`. last access: July 7, 2012.

[3] Qemu open source processor emulator. `http://www.qemu.org`. last access: July 7, 2012.

[4] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine applications. Technical report, University of California at Santa Barbara, 1999.

[5] Advanced Micro Devices, Inc. *AMD64 Architecture Programmer's Manual*, 2011.

[6] G. Anthes. Security in the cloud. *Communications of the ACM*, 53(11):16–18, Nov. 2010.

[7] ARM Ltd. *Cortex-A15 MPCore Technical Reference Manual*, 2012.

[8] F. Baiardi, D. Maggiari, D. Sgandurra, and F. Tamberi. Transparent process monitoring in a virtual environment. *Electronic Notes in Theoretical Computer Science*, 236:85–100, Apr. 2009.

[9] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *ACM SIGOPS Operating Systems Review*, 37(5):164–177, Oct. 2003.

[10] U. Bayer, C. Kruegel, and E. Kirda. TTAnalyze: A tool for analyzing malware. In *Proceedings of the Conference of the European Institute for Computer Antivirus Research*, Hamburg, Germany, 2006.

[11] D. Beck, B. Vo, and C. Verbowski. Detecting stealth software with strider ghostbuster. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 368–377, Washington, DC, USA, 2005. IEEE.

[12] M. Carbone, W. Cui, L. Lu, W. Lee, M. Peinado, and X. Jiang. Mapping kernel objects to enable systematic integrity checking. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 555–565, New York, NY, USA, 2009. ACM.

[13] C.-C. Chang and C.-J. Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[14] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems*, pages 133–138, Washington, DC, USA, 2001. IEEE.

[15] M. Christodorescu, R. Sailer, D. L. Schales, D. Sgandurra, and D. Zamboni. Cloud security is not (just) virtualization security: a short paper. In *Proceedings of the ACM Workshop on Cloud Computing Security*, pages 97–102, New York, NY, USA, 2009. ACM.

[16] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers*. O'Reilly, Sebastopol, CA, USA, 2005.

[17] A. Dinaburg, P. Royal, M. Sharif, and W. Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 51–62, New York, NY, USA, 2008. ACM.

[18] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 297–312, Washington, DC, USA, 2011. IEEE.

[19] B. Dolan-Gavitt, B. D. Payne, and W. Lee. Leveraging forensic tools for virtual machine introspection. Technical report, Georgia Institute of Technology, 2011.

[20] A. Fattori, R. Paleari, L. Martignoni, and M. Monga. Dynamic and transparent analysis of commodity production systems. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 417–426, New York, NY, USA, 2010. ACM.

[21] S. Forrest, S. Hofmeyr, and A. Somayaji. The evolution of system-call monitoring. In *Proceedings of the Annual Computer Security Applications Conference*, pages 418–430, Washington, DC, USA, 2008. IEEE.

[22] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 120–128, Washington, DC, USA, 1996. IEEE.

[23] T. Garfinkel. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 163–176, 2003.

[24] T. Garfinkel, K. Adams, A. Warfield, and J. Franklin. Compatibility is not transparency: VMM detection myths and realities. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems*, pages 1–6, Berkeley, CA, USA, 2007. USENIX.

[25] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the Network and Distributed System Security Symposium*, 2004.

[26] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of the Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[27] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications confining the wily hacker. In *Proceedings of the USENIX Security Symposium*, pages 1–13, Berkeley, CA, USA, 1996. USENIX.

[28] R. P. Goldberg. *Architectural Principles for Virtual Computer Systems.* PhD thesis, Harvard University, Cambridge, MA, 1972.

[29] R. P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, pages 34–46, 1974.

[30] M. Grace, Z. Wang, D. Srinivasan, J. Li, X. Jiang, Z. Liang, and S. Liakh. Transparent protection of commodity os kernels using hardware virtualization. In *Proceedings of the International Conference on Security and Privacy in Communications Networks*, pages 162–180, Berlin, Heidelberg, 2010. Springer.

[31] Z. Gu, Z. Deng, D. Xu, and X. Jiang. Process implanting: A new active introspection framework for virtualization. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 147–156, Washington, DC, USA, 2011. IEEE.

[32] B. Hay and K. Nance. Forensics examination of volatile system data using virtual introspection. *ACM SIGOPS Operating Systems Review*, 42(3):74–82, 2008.

[33] R. Herbrich. *Learning Kernel Classifiers: Theory and Algorithms.* MIT Press, Cambridge, MA, USA, 2001.

[34] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[35] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2012.

[36] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through VMM-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 128–138, New York, NY, USA, 2007. ACM.

[37] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Berkeley, CA, USA, 2006. USENIX.

[38] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. VMM-based hidden process detection and identification using Lycosid. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 91–100, New York, NY, USA, 2008. ACM.

[39] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting past and present intrusions through vulnerability-specific predicates. *ACM SIGOPS Operating Systems Review*, 39(5):91–104, Dec. 2005.

[40] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In *Proceedings of the Linux Symposium*, pages 225–230, Ottawa, ON, Canada, 2007.

[41] J. Z. Kolter and M. A. Maloof. Learning to detect and classify malicious executables in the wild. *Journal of Machine Learning Research*, 7:2721–2744, Dec. 2006.

[42] A. P. Kosoresow and S. A. Hofmeyr. Intrusion detection via system call traces. *IEEE Software*, 14(5):35–42, 1997.

[43] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. Accessminer: using system-centric models for malware protection. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 399–412, New York, NY, USA, 2010. ACM.

[44] Y. Liao and V. R. Vemuri. Using text categorization techniques for intrusion detection. In *Proceedings of the USENIX Security Symposium*, pages 51–59, Berkeley, CA, USA, 2002. USENIX.

[45] H.-T. Lin, C.-J. Lin, and R. C. Weng. A note on platt's probabilistic outputs for support vector machines. *Machine Learning*, 68(3):267–276, Oct. 2007.

[46] L. Litty, H. A. Lagar-Cavilla, and D. Lie. Hypervisor support for identifying covertly executing binaries. In *Proceedings of the USENIX Security Symposium*, pages 243–258, Berkeley, CA, USA, 2008. USENIX.

[47] L. Litty and D. Lie. Manitou: a layer-below approach to fighting malware. In *Proceedings of the Workshop on Architectural and System Support for Improving Software Dependability*, pages 6–11, New York, NY, USA, 2006. ACM.

[48] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. *Journal of Machine Learning Research*, 2:419–444, March 2002.

[49] S. Loh. Inside windows ce api calls. `http://blogs.msdn.com/b/ce_base/archive/2006/02/02/inside-windows-ce-api-calls.aspx`, Feb. 2006. last access: Sep. 25, 2012.

[50] F. Lombardi and R. D. Pietro. Secure virtualization for cloud computing. *Journal of Network and Computer Applications*, 34(4):1113 – 1122, 2011.

[51] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[52] E. G. Mallach. On the relationship between virtual machines and emulators. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 117–126, New York, NY, USA, 1973. ACM.

[53] L. Martignoni, A. Fattori, R. Paleari, and L. Cavallaro. Live and trustworthy forensic analysis of commodity production systems. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 297–316, Berlin, Heidelberg, 2010. Springer.

[54] K. Nance, M. Bishop, and B. Hay. Virtual machine introspection: Observation or interference? *IEEE Security & Privacy*, 6(5):32 –37, sept.-oct. 2008.

[55] P. Neira-Ayuso, R. M. Gasca, and L. Lefevre. Communicating between the kernel and user-space in linux using netlink sockets. *Software Practice and Experience*, 40(9):797–810, Aug. 2010.

[56] B. D. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of the Annual Computer Security Applications Conference*, pages 385–397, Washington, DC, USA, 2007. IEEE.

[57] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 233–247, Washington, DC, USA, 2008. IEEE.

[58] J. C. Platt. *Probabilistic Outputs for Support Vector Machines and Comparisons to Regularized Likelihood Methods*, chapter 5, pages 61–74. MIT Press, Cambridge, MA, USA, 2000.

[59] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

[60] N. Provos. Improving host security with system call policies. In *Proceedings of the USENIX Security Symposium*, pages 18–18, Berkeley, CA, USA, 2003. USENIX.

[61] N. A. Quynh and K. Suzaki. Xenprobes, a lightweight user-space probing framework for xen virtual machine. In *Proceedings of the USENIX Annual Technical Conference*, pages 2:1–2:14, Berkeley, CA, USA, 2007. USENIX.

[62] J. Rhee, R. Riley, D. Xu, and X. Jiang. Defeating dynamic data kernel rootkit attacks via vmm-based guest-transparent monitoring. In *Proceedings of the International Conference on Availability, Reliability and Security*, pages 74–81, Washington DC, USA, 2009. IEEE.

[63] J. Rhee and D. Xu. LiveDM: Temporal mapping of dynamic kernel memory for dynamic kernel malware analysis and debugging. Technical report, Purdue University, 2010.

[64] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *Proceedings of the Conference on Detection of Intrusions and Malware and Vulnerability Assessment*, pages 108–125, Berlin, Heidelberg, 2008. Springer.

[65] K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. Technical report, Berlin Institute of Technology, 2009.

[66] R. Riley, X. Jiang, and D. Xu. Guest-transparent prevention of kernel rootkits with vmm-based memory shadowing. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 1–20, Berlin, Heidelberg, 2008. Springer.

[67] M. Rosenblum and T. Garfinkel. Virtual machine monitors: Current technology and future trends. *IEEE Computer*, 38(5):39–47, May 2005.

[68] J. Rutkowska. Subverting vista kernel for fun and profit. /urlhttp://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Rutkowska.pdf, 2006. last access: July 30, 2012.

[69] C. Schneider, J. Pfoh, and C. Eckert. A universal semantic bridge for virtual machine introspection. In *Proceedings of the International Conference on Information Systems Security*, pages 370–373, Berlin, Heidelberg, 2011. Springer.

[70] C. Schneider, J. Pfoh, and C. Eckert. Bridging the semantic gap through static code analysis. In *Proceedings of the European Workshop on System Security*, New York, NY, USA, 2012. ACM.

[71] B. Schölkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond.* MIT Press, Cambridge, MA, USA, 2001.

[72] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo. Data mining methods for detection of new malicious executables. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 38–49, Washington DC, USA, 2001. IEEE.

[73] M. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-VM monitoring using hardware virtualization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 477–487, New York, NY, USA, 2009. ACM.

[74] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, Feb. 1993.

[75] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[76] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu. Process out-grafting: an efficient "out-of-VM" approach for fine-grained process execution monitoring. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 363–374, New York, NY, USA, 2011. ACM.

[77] A. Srivastava and J. Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 39–58, Berlin, Heidelberg, 2008. Springer.

[78] A. Srivastava and J. Giffin. Automatic discovery of parasitic malware. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 97–117, Berlin, Heidelberg, 2010. Springer.

[79] A. Srivastava, A. Lanzi, and J. Giffin. System call api obfuscation (extended abstract). In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 421–422, Berlin, Heidelberg, 2008. Springer.

[80] The Honeynet Project. Know your enemy: Sebek. Technical report, The Honeynet Project, 2003.

[81] The Honeynet Project. *Know Your Enemy: Learning about Security Threats.* Addison-Wesley Professional, Boston, MA, USA, 2004.

[82] S. Vogl and C. Eckert. Using hardware performance events for instruction-level monitoring on the x86 architecture. In *Proceedings of the European Workshop on System Security*, New York, NY, USA, 2012. ACM.

[83] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–168, Washington, DC, USA, 2001. IEEE.

[84] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 255–264, New York, NY, USA, 2002. ACM.

[85] X. Wang, W. Yu, A. Champion, X. Fu, and D. Xuan. Detecting worms via mining dynamic program execution. In *Proceedings of the International Conference on Security and Privacy in Communications Networks*, 2007.

[86] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 545–554, New York, NY, USA, 2009. ACM.

[87] C. Warrender, S. Forrest, and B. A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 133–145, Washington DC, USA, 1999. IEEE.

[88] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the USENIX Annual Technical Conference*, Berkeley, CA, USA, 2002. USENIX.

[89] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and performance in the denali isolation kernel. *ACM SIGOPS Operating Systems Review*, 36(SI):195–209, Dec. 2002.

[90] C. Willems, T. Holz, and F. Freiling. Toward automated dynamic malware analysis using CWSandbox. *IEEE Security & Privacy*, 5(2):32–39, Mar. 2007.

[91] G. Xiang, H. Jin, D. Zou, X. Zhang, S. Wen, and F. Zhao. VMDriver: A driver-based monitoring mechanism for virtualization. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 72–81, Washington, DC, USA, 2010. IEEE.

[92] H. Xiao and T. Stibor. A supervised topic transition model for detecting malicious system call sequences. In *Proceedings of the Workshop on Knowledge Discovery, Modeling and Simulation*, New York, NY, USA, 2011. ACM.